DRAFT SF 298

1. Report Date (dd-mm-yy) April 1995	2. Report Type	3. Dates covered (from... to)
4. Title & subtitle Cleanroom Pamphlet		5a. Contract or Grant # 5b. Program Element #
6. Author(s)		5c. Project # 5d. Task # 5e. Work Unit #
7. Performing Organization Name & Address		8. Performing Organization Report #
9. Sponsoring/Monitoring Agency Name & Address Software Technology Support Center OO-ALC/TISE 7278 4th Street Hill AFB, UT 84056-5205		10. Monitor Acronym 11. Monitor Report #

12. Distribution/Availability Statement
Distribution Statement A: Approved for public release, distribution is unlimited.

13. Supplementary Notes

14. Abstract

19970516 148

DTIC QUALITY INSPECTED 4

15. Subject Terms

Security Classification of			19. Limitation of Abstract	20. # of Pages	21. Responsible Person (Name and Telephone #)
16. Report Unclass	17. Abstract Unclass	18. This Page Unclass			Randy Wright (801) 777-9732

STSC

Cleanroom Pamphlet

April 1995

CLEANROOM PAMPHLET

"Software products with reliable operating characteristics have been notoriously difficult to develop. The Cleanroom method introduces sound software engineering into the development cycle and provides the quality control that's essential for product success." [1]

This pamphlet is part of the ongoing effort by the Software Technology Support Center (STSC) in assisting Air Force organizations to identify, evaluate and adopt technologies that will improve: (1) the quality of their products, (2) their efficiency in producing those products, and (3) their ability to accurately predict the cost and schedule of their delivery. The publication of this pamphlet makes no implications that the STSC is endorsing the Cleanroom process over any other technology. The STSC just feels that this is a technology that is worth looking into as a method for improving software quality, and possibly increasing productivity. The Cleanroom process may be a viable method for some organizations and may not work for others. Cleanroom, just like any other technology, is something that needs to be evaluated by the individual organizations.

The STSC's Software Quality Engineering (SQE) Team is responsible for providing products and services pertaining to software quality. Members of the SQE Team are: Bryce Ragland, Johnnie Henderson (LORAL), and Mark Dawood (SAIC).

The following items are included in the Cleanroom Pamphlet:

[1] Michael Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, Inc. New York, 1994 (Back Cover)

Cleanroom Software Engineering: Management Overview

Richard C. Linger
Software Engineering Institute

Cleanroom Software Engineering: Management Overview

Richard C. Linger

Developing Software Under Statistical Quality Control

Cleanroom software engineering is a theory-based, team-oriented process for on-schedule development and certification of ultra-high-reliability software systems with improved productivity under statistical quality control [2,3]. The Cleanroom name is borrowed from hardware Cleanrooms, with their emphasis on prevention of errors through engineering discipline, rather than error removal. Cleanroom combines rigorous methods of software specification, design, correctness verification, and statistical quality certification in a new life cycle model based on incremental development.

You can use the Cleanroom process and supporting technologies to develop software systems that approach zero defects and have scientifically certified reliability for operational field use.

In contrast to traditional development approaches, in the Cleanroom process you embed software development and testing within a formal statistical quality control process. In such a process, software engineering is required to create software that approaches zero defects and can enter system testing directly [1]. Then statistical usage-based testing is used to provide statistical inferences about the reliability of the software. This systematic process of assessing and controlling software quality during development permits you to certify product reliability at delivery, based on a complete public record of the testing and all engineering change activity required to deliver acceptable software.

The significance of a process under statistical quality control is well illustrated by modern manufacturing techniques where the sampling of output is directly fed back into the process to control quality. Once the discipline of statistical quality control is in place, management has objective visibility into the software development process and can control process changes to control product quality.

Key ingredients of the Cleanroom process are a new development life cycle and independent quality assessment through statistical testing. You begin the development life cycle with a specification that not only defines functional requirements, but also identifies statistical usage of the software and a nested sequence of user-function subsets that can be released and tested as increments which accumulate into the final system. Rigorous software engineering methods provide design and correctness verification techniques to create provably correct software. Correctness verification by software engineering teams has proven to be a powerful and effective process for approaching zero defects prior to any execution of the software.

You place software under engineering change control from first execution on, and all execution is controlled by an independent certification team that uses statistical testing methods to evaluate software quality. Traditional structural or coverage tests, no matter how carefully selected or how comprehensive their test plans, provide only elaborate anecdotes of quality, with no basis for scientific statistical extrapolation to operational environments. However, statistical testing in a quality control process results in objective quality certification of software at acceptance.

Cleanroom Results

The Cleanroom process has been developed, reduced to practice, and demonstrated in development of a variety of software systems in a various languages and environments. Published results of projects totaling over a million lines of code carried out by IBM, STARS, NASA, and other organizations [1,2] have shown substantial improvements over traditional results, as the following examples illustrate:

Quality. Improvements of 10-20X and more over baseline performance have been achieved. Some Cleanroom-developed systems have experienced no errors whatsoever in field use. For example, IBM developed an embedded, real-time, bus architecture, multiple-processor device controller product that experienced no errors in two years use at over 300 customer locations.

Productivity. Gains of 1.5-5X over baseline performance have been reported. For example, an Ericsson Telecom project to develop a 374 KLOC operating system reported gains of 1.7X in development productivity, 1.6X in testing productivity, and an IBM project to develop a network management and outage avoidance product reported 2X improvement in development productivity.

Life cycle costs. Dramatic reductions have been achieved due to sharp decreases in error correction and maintenance costs over the life of a product. For example, IBM developed a COBOL structuring product that experienced just seven minor errors in the first three years of field use, all simple fixes, and required a small fraction of the maintenance budget associated with products of similar complexity.

Return on investment. Experience shows that Cleanroom adoption costs can be recovered on the first project. For example, an 11 to 1 ROI was reported by the Picatinny Arsenal STARS Cleanroom project, with the investment covering all Cleanroom training and consultation costs.

You can experience other benefits of the Cleanroom process that are more difficult to quantify, but are real nonetheless. For example, Cleanroom statistical testing provides you with scientific measures of product quality for the first time, permitting objective decision-making on whether and when to stop testing and release products. It also provides scientific projections of quality in field use, the only known method for doing this. In addition, substantial increases in the job satisfaction of Cleanroom teams have been reported [8].

You can apply the Cleanroom process to development of new systems and maintenance, evolution, and re-engineering of existing systems. It is language, environment, and subject-matter independent, and can be used to develop and evolve a variety of systems, including real-time, embedded, host, distributed, workstation, client-server, and microcode systems. Cleanroom is compatible with prototyping and promotes reuse through precise definition of component functional semantics and certification of component reliability.

Cleanroom is compatible with and supports performance at SEI Capability Maturity Model levels 2 through 5 [7]. Organizations at CMM level 1 may wish to achieve level 2 before embarking on Cleanroom operations.

Cleanroom Technologies

The Cleanroom process incorporates technologies for management, development, and testing, as follows:

Incremental Development. Cleanroom management is based on incremental development and certification of a pipeline of user-function increments that accumulate into the final product. System integration is top-down and continual, with system functionality growing through addition of successive increments. Incremental development enables early and continual assessment of product quality and user feedback, and facilitates improvements as development progresses. The incremental approach permits controlled, stepwise integration of components, avoiding the risky, last-minute integration often experienced in traditional development.

Rigorous Specification and Design. The development team produces software approaching zero defects through use of a rigorous stepwise refinement and verification process for specification and design using object-based Box Structure technology [6]. Box Structures permit precise definition of required user function and system object architecture, and scale up to maintain intellectual control in large system development. A key concept in Box Structures is referential transparency, whereby the subspecifications for successive object refinements are developed, connected and verified in a coherent structure prior to their independent elaboration. In Cleanroom, correctness is built in, not tested in.

Correctness Verification. All Cleanroom-developed software is subject to rigorous correctness verification by the development team prior to release to the certification test team. A practical and powerful process, verification permits development teams to completely verify the correctness of software with respect to specifications. A Correctness Theorem defines conditions to be met for achieving zero-defect software [4]. These conditions are verified in mental/verbal proofs of correctness in development team reviews. Even though programs of any size contain a virtually infinite number of paths, the theorem reduces verification to a finite number of checks and ensures that all software logic is completely verified in all possible circumstances of use. The verification step is extremely powerful in eliminating defects, and is a major factor in the dramatic quality improvements experienced by Cleanroom teams.

Statistical Quality Certification. The objective of the certification test team is to provide scientific certification of software reliability, not to test quality in, an impossible task. Following correctness verification, software increments are placed under engineering change control and undergo first execution. Statistical usage testing is carried out to produce scientifically valid measures of software quality and reliability [5]. The statistical usage approach tests software the way users intend to use it. Test cases are generated based on usage probability distributions that model anticipated software use in all possible circumstances, including unusual and stress situations. Usage distributions can be defined in Markov models that permit substantial management analysis and simulation of test operations, as well as automatic test case generation [9]. Objective statistical measures of software reliability, such as Mean Time To Failure (MTTF), are computed based on test results for informed management decision-making. Because statistical usage testing is biased toward detection of more serious, high-frequency errors first, it is more effective at improving software reliability in less time than traditional testing techniques.

Cleanroom Application

You can apply Cleanroom practices in the following environments:

New Systems. You can use the Cleanroom process to provide a coherent management and technical framework for on-schedule development under intellectual control. Incremental development provides for early quality assessment and user feedback on system function, and avoids the risk associated with late component integration in waterfall-based developments.

Existing Systems. You can develop modifications and extensions to existing systems using Cleanroom technology. In addition, problem-prone modules in existing systems can be re-engineered to Cleanroom quality through use of design abstraction and correctness verification techniques.

Cleanroom Acquisition. You can integrate the Cleanroom process into acquisition practices in terms of required project processes and deliverables. For example, an incremental development process can be required, with incremental deliverables associated with rigorous software specification, design, correctness verification, and statistical reliability certification. Required software MTTF and other statistical measures can be specified as prerequisites for software acceptance at delivery.

Phased Introduction of Cleanroom

You can introduce the Cleanroom process into an organization in a staged manner. A successful strategy has been to start with pilot projects and teams for Cleanroom development. Success with these projects provides incentives for widespread adoption. As use of the Cleanroom process grows, experienced members of early teams can become leaders of new teams.

Within a project, a phased introduction of Cleanroom process and technology elements is possible [1]. As experience accumulates, you can adopt successive elements of Cleanroom in a staged implementation.

References

1. Hausler, P. A., R. C. Linger, and C. T. Trammell, "Adopting Cleanroom Software Engineering with a Phased Approach," *IBM Systems Journal*, March, 1994.

2. Linger, R. C., "Cleanroom Software Engineering for Zero-Defect Software," *Proceedings of 15th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 1993.

3. Linger, R. C., “Cleanroom Process Model,” *IEEE Software*, IEEE Computer Society Press, Los Alamitos, CA, Vol. 11, No. 2, March, 1994.

4. Linger, R. C., and H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.

5. Mills, H. D., “Certifying the Correctness of Software,” *Proceedings of 25th Hawaii International Conference on System Sciences,* IEEE Computer Society Press, January, 1992, pp. 373-381.

6. Mills, H. D., R. C. Linger, and A. R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, New York, 1986.

7. Paulk, M. C., et al., *Capability Maturity Model for Software, Version 1.1* (CMU/SEI-93-TR-25 ADA263432). Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University, 1993.

8. Sherer, S. W., P. G. Arnold, and A. Kouchakdjian, "Experience Using Cleanroom Software Engineering in the US Army," *Proceedings of Second Annual European Industrial Symposium on Cleanroom Software Engineering*, Q-Labs AB, IDEON Research Park, S-223 70 Lund, Sweden.

9. Whittaker, J. A. and M. G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering*, IEEE Computer society Press, Los Alamitos, CA, Vol. 20, No. 4, October, 1994.

Reprinted From:

IBM
Systems Journal
Vol. 33, No. 1, 1994

Adopting Cleanroom software engineering with a phased approach

by P. A. Hausler
R. C. Linger
C. J. Trammell

Adopting Cleanroom software engineering with a phased approach

by P. A. Hausler
R. C. Linger
C. J. Trammell

Cleanroom software engineering is a theory-based, team-oriented engineering process for developing very high quality software under statistical quality control. The Cleanroom process combines formal methods of object-based box structure specification and design, function-theoretic correctness verification, and statistical usage testing for reliability certification to produce software approaching zero defects. Management of the Cleanroom process is based on a life cycle of development and certification of a pipeline of user-function increments that accumulate into the final product. Teams in IBM and other organizations that use the process are achieving remarkable quality results with high productivity. A phased implementation of the Cleanroom process enables quality and productivity improvements with an increased control of change. An introductory implementation involves the application of Cleanroom principles without the full formality of the process; full implementation involves the comprehensive use of formal Cleanroom methods; and advanced implementation optimizes the process through additional formal methods, reuse, and continual improvement. The AOEXPERT/MVS™ project, the largest IBM Cleanroom effort to date, successfully applied an introductory level of implementation. This paper presents both the implementation strategy and the project results.

Zero or near-zero defect software may seem like an impossible goal. After all, the experience in the first generation of software development has reinforced the seeming inevitability of errors and persistence of human fallibility. Today, however, a new reality in software development belies the first-generation experience. Cleanroom software engineering teams are able to develop software at a level of quality and reliability that would have seemed impossible a few years ago, and are doing so with high productivity.

Cleanroom software engineering is a managerial and technical process for the development of software approaching zero defects with certified reliability.[1,2] The Cleanroom process spans the entire software life cycle; it provides a complete discipline within which software teams can plan, specify, design, verify, code, test, and certify software. The Cleanroom approach treats software development as an engineering process based on mathematical foundations, rather than as a trial-and-error programming process,[3-7] and is intended to produce software with error-free designs and failure-free executions.

In traditional, craft-based software development, errors were accepted as inevitable, and programmers were encouraged to get software into testing quickly in order to begin debugging. Programs were subjected to unit testing and debugging by their authors, then integrated into components, subsystems, and systems for more debugging.

Product use by customers resulted in still more debugging to correct errors discovered in operational use. The most virulent errors were often the result of fixes to other errors,[8] and it was not unusual for software products to reach a steady-state error population, with new errors introduced as fast as old ones were fixed. Today, however, craft-based processes that depend on testing and debugging to improve reliability are understood to be inefficient and ineffective. Experience has shown that craft-based processes often fail to achieve the level of reliability essential to a society dependent on software for the conduct of human affairs.

In the Cleanroom process, correctness is built into the software by development teams through a rigorous engineering process of specification, design, and verification. The more powerful process of team correctness verification replaces unit testing and debugging, and software enters system testing directly, with no execution by development teams. All errors are accounted for from first execution on, with no private unit testing necessary or permitted. Experience shows that Cleanroom software typically enters system testing approaching zero defects and occasionally no defects are found in all testing.

Certification (test) teams are not responsible for "testing in" quality, which is an impossible task, but rather for certifying the quality of software with respect to its specification. Certification is performed by statistical usage testing that produces objective assessments of product quality. Errors, if any, found in testing are returned to the development team for correction. If the quality is not acceptable, the software is removed from testing and returned to the development team for reverification.

The process of Cleanroom development and certification is carried out in an incremental manner. System functionality grows with the addition of successive code increments in a stepwise integration process. When the final increment is added, the system is complete. Because successive increments are elaborating the top-down design of increments already in execution, interface and design errors are rare.

This paper describes key Cleanroom technologies and summarizes quality results achieved by Cleanroom teams. It presents a phased approach to Cleanroom implementation based on the software maturity level of an organization, and summarizes the results of a substantial IBM Cleanroom project (AOEXPERT/MVS*) that successfully applied a phased approach.

Cleanroom perspectives

The Cleanroom software engineering process evolved from concepts developed and demonstrated over the past 15 years by Harlan Mills and colleagues.[3-5,9] Cleanroom practices such as stepwise refinement of procedure and object hierarchies, team verification of correctness, and statistical usage testing, have been successfully applied in commercial and governmental software projects over the past decade. Such practices may not be the rule in software development today, but their use is growing as evidence of their value continues to accumulate. In many cases, software organizations considering a transition to the Cleanroom process have operational practices in place, such as incremental development, structured programming, and team reviews, that support Cleanroom concepts. There are only a few key concepts that must be understood and accepted in a transition to the Cleanroom approach.[10]

Practice based on theory. To be effective, any engineering discipline must be based on sound theoretical foundations. Cleanroom specification, design, and correctness verification practices are based on function theory, whereby programs are treated as rules for mathematical functions subject to stepwise refinement and verification.[4,5] Cleanroom testing and quality certification practices are based on statistical theory, whereby program executions are treated as populations subject to usage-based, stochastic sampling in formal statistical designs.[3,6,11] These theoretical foundations form the basis of a comprehensive engineering process that has been reduced to practice for commercial software development. A growing number of successful, real-world Cleanroom projects have demonstrated the practicality of these methods.

Experienced Cleanroom practitioners and educators have developed comprehensive technology transfer programs based on readily teachable, time-efficient approaches to such Cleanroom technologies as correctness verification and statistical testing. New practitioners will find that processes and tools exist that make the use of these Cleanroom methods highly practical.[12]

Right the first time. A primary objective of the Cleanroom process is to prevent errors, rather than accepting and accommodating errors through institutionalized debugging and rework. For this reason, Cleanroom development teams do not unit test and debug their code. Instead, they rely on rigorous methods of specification and design combined with team correctness verification. These Cleanroom development practices, based on mathematical foundations, yield quality approaching zero defects prior to first execution by certification teams. The purpose of testing in Cleanroom is the certification of software quality with respect to specifications, not the attempt to "debug in" quality.

Management understanding and acceptance of this essential point—that quality will be achieved by design and verification rather than by testing—must be reflected in the development schedule. Time spent in specification and design phases of a Cleanroom development is greater than in traditional projects. Time spent in testing, however, is likely to be less than traditionally required. The manager who wanted to start coding immediately because of the large amount of debugging expected was usually right, but would have difficulty becoming part of a Cleanroom team.

Quality costs less. A principal justification for the Cleanroom process is that built-in quality lowers the overall cost to produce and maintain a product. The exponential growth in the cost of error correction in successive life-cycle phases is well known. Errors found in operational use by customers are typically several orders of magnitude more costly to correct than errors found in the specification phase.[13] The Cleanroom name, taken from the semiconductor industry where a literal cleanroom exists to prevent introduction of defects during hardware fabrication, is a metaphor that reflects this understanding of the cost-effectiveness of error prevention. In the Cleanroom process, incremental development and extensive team review and verification permit errors to be detected as early as possible in the life cycle. By reducing the cost of errors during development and the incidence of failures during operation, the overall life-cycle cost of Cleanroom software can be expected to be far lower than industry averages. For example, the IBM COBOL Structuring Facility product, developed using Cleanroom techniques, has required only a small fraction of its maintenance budget to be consumed during years of field use.

Cleanroom project schedules have equaled or improved upon traditional development schedules.[14–16] In fact, productivity improvements of factors ranging from one and one-half to five over traditional practices have been observed.[15–18] Experienced Cleanroom teams become remarkably efficient at writing clear specifications, simplifying and restricting designs to easily verifiable patterns, and performing correctness verification. Cleanroom is not a more time-consuming development process, but it does place greater emphasis on design and verification to avoid waste of resources in debugging and rework.

A primary objective of the Cleanroom process is to prevent errors.

Cleanroom quality results

As summarized in Table 1, first-time Cleanroom teams in IBM and other industrial and governmental organizations have reported data on close to a million lines of Cleanroom-developed software. The code exhibits a weighted average of 2.3 errors per thousand lines of code (errors/KLOC) in testing.[2,15–19] This error rate represents all errors found in all testing, measured from first-ever execution through test completion. That is, it is a measure of residual errors remaining following correctness verification by development teams, who do not execute the software. The projects represent a variety of environments, including batch, distributed, cooperative, and real-time systems and system parts, and a variety of languages, including microcode, C, C++, JOVIAL, FORTRAN, and PL/I.

Traditionally developed software does not undergo correctness verification, but rather enters unit testing and debugging directly, followed by more debugging in function and system testing

Table 1 Cleanroom project results

Year	Project	Quality and Productivity
1987	IBM Flight Control: Helicopter Avionics System Component 33 KLOC (JOVIAL)	• Certification testing failure rate: 2.3 errors/KLOC • Error-fix reduced 5X • Completed ahead of schedule
1988	IBM Cobol Structuring Facility: Product for automatically restructuring COBOL programs 85 KLOC (PL/I)	• IBM's first Cleanroom product • Certification testing failure rate: 3.4 errors/KLOC • Productivity 740 LOC/PM, 5X improvement • 7 errors in first 3 years of use; all simple fixes
1989	NASA Satellite Control Project 1 40 KLOC (FORTRAN)	• Certification testing failure rate: 4.5 errors/KLOC • 50% improvement in quality • Productivity 780 LOC/PM • 80% improvement in productivity
1990	Martin Marietta: Automated documentation system 1.8 KLOC (FOXBASE)	• First compilation: no errors found • Certification testing failure rate: 0.0 errors/KLOC (no errors found)
1991	IBM System Software First increment 0.6 KLOC (C)	• First compilation: no errors found • Certification testing failure rate: 0.0 errors/KLOC (no errors found)
1991	IBM AOEXPERT/MVS™ Product 107 KLOC (mixed languages)	• Testing failure rate: 2.6 errors/KLOC • Productivity 486 LOC/PM • No operational errors from Beta test sites
1991	IBM Language Product First increment 21.9 KLOC (PL/X)	• Testing failure rate: 2.1 errors/KLOC
1991	IBM Image Product Component 3.5 KLOC (C)	• First compilation: 5 syntax errors • Certification testing failure rate: 0.9 errors/KLOC
1992	IBM Printer Application First increment 6.7 KLOC (C)	• Certification testing failure rate: 5.1 errors/KLOC
1992	IBM Knowledge Based System Application 17.8 KLOC (TIRS™)	• Testing failure rate: 3.5 errors/KLOC
1992	NASA Satellite Control Projects 2 and 3 170 KLOC (FORTRAN)	• Testing failure rate: 4.2 errors/KLOC
1993	University of Tennessee: Cleanroom tool 20 KLOC (C)	• Certification testing failure rate: 6.1 errors/KLOC
1993	IBM 3490E Tape Drive 86 KLOC (C)	• Certification testing failure rate: 1.2 errors/KLOC
1993	IBM Database Transaction Processor First increment 21.5 KLOC (JOVIAL)	• Testing failure rate: 2.4 errors/KLOC • No design errors, all simple fixes
1993	IBM LAN Software First increment 4.8 KLOC (C)	• Testing failure rate: 0.8 errors/KLOC
1993	IBM Workstation Application Component 3.0 KLOC (JOVIAL)	• Testing failure rate: 4.1 errors/KLOC
1993	Ericsson Telecom AB Switching Computer OS32 Operating System 350 KLOC (PLEX, C)	• Testing failure rate: 1 error/KLOC • 70% improvement in development productivity • 100% improvement in testing productivity

NOTE: All testing failure rates are measured from first-ever execution.

KEY: KLOC = thousand lines of code
PM = person month
X = (mathematical) times

following. Measured from first execution, traditional software typically exhibits 25 to 35 or more errors per thousand lines of code.[20] First-time Cleanroom development teams can produce software with quality levels at test entry at least an order of magnitude better than traditionally developed software. The following summaries of three selected projects from Table 1 illustrate the results achieved.

IBM COBOL Structuring Facility. The COBOL Structuring Facility, which consisted of 85 KLOC of PL/I code, was the first Cleanroom product in IBM. It employs proprietary, graph-theoretic algorithms to automatically transform unstructured COBOL programs into a functionally equivalent, structured form for improved maintainability. Relentless design simplification in the Cleanroom process often results in systems that are small for their functionality. For example, the Cleanroom-developed prototype of the COBOL Structuring Facility, independently estimated at 100 KLOC, was developed using just 20 KLOC.

Comparable to a COBOL compiler in complexity, the product experienced 3.4 errors/KLOC in all statistical testing, measured from the first execution. Six months of intensive beta testing at a major aerospace corporation resulted in no functional equivalence errors ever found.[21] Just seven minor errors were reported in the first three years of field use, requiring only a small fraction of the maintenance budget associated with traditionally developed products of similar size and complexity. The product was developed and certified by a team averaging six members, with productivity five times the IBM averages.[16]

IBM 3490E tape drive. The 3490E tape drive is a real-time, embedded software system developed by a five-person team in three increments of C design with a code total of 86 KLOC. It provides high-performance tape cartridge support through a multiple processor bus architecture that processes multiple real-time input and output data streams. The product experienced 1.2 errors/KLOC in all statistical testing. To meet an urgent business need, the third increment was shipped straight from development to the hardware and software integration team with no testing whatsoever. Customer evaluation testing with live data by the integration team resulted in no errors being found.

In a comparison experiment, the project team subjected a selected module to both unit testing and correctness verification. Development of execution scaffolding, definition and execution of test cases, and checking of results required one-and one-half person-weeks of effort and resulted in the detection of seven errors. Correctness verification of the same program by the development team required one and one-half hours, and resulted in the detection of the same seven errors, plus three additional errors.[1]

Ericsson OS32 operating system. Ellemtel Telecommunications Systems Laboratories is completing a 350 KLOC operating system for a new family of switching computers for Ericsson Telecom AB. The code is written in PLEX and C. The 73-person, 33-month Cleanroom project experienced productivity improvements of 70 percent and 100 percent in development and testing, respectively, and the product averaged under one error/KLOC in all testing. Project management reported that an average of less than one person-hour was required to detect an error in team reviews, compared to an average of 17.5 person-hours to detect an error in testing. The project allocated two days per week to prepare and conduct team reviews. The product team was honored by Ericsson as the single project that had contributed the most to the company in 1993.[18]

Cleanroom technologies

In the Cleanroom process, the objective of the development team is to deliver software to the test team that approaches zero defects; the objective of the test team is to scientifically certify the quality of software, not to attempt to "test in" quality. These objectives are achieved through management and technical practices based on the technologies of incremental development, box structure specification and design, correctness verification, and statistical quality certification.

Incremental development. Management planning and control in Cleanroom is based on development and certification of a *pipeline of increments* that represent operational user function, accumulate top-down into the final product, and execute in the system environment.[22] Following specification of required external system behavior, an incremental development plan is created to define schedules, resources, and functional content of a series of code increments to be developed and certified. The initial increment contains stubs (small placeholder programs) that stand in for later increments and permit early execution of the code. The ultimate functionality of the code that will replace the stubs is fully defined in subspecifications for team verification of each increment prior to testing. As incremental development progresses, stubs are replaced by corresponding code increments, possibly containing stubs of their own, in a stepwise system integration process. When the final increment is integrated, the system is complete and no stubs remain.

When the final increment is integrated, the system is complete.

As each increment is integrated, the evolving system of increments undergoes a new step in statistical usage testing for quality certification. Statistical measures of quality provide feedback for reinforcement or improvement of the development process as necessary. Early increments can serve as system prototypes, providing an opportunity to elicit feedback from customers to validate requirements and functionality. As inevitable changes occur, incremental development provides a framework for revising schedules, resources, and function, and permits changes to be incorporated in a systematic manner.

Box structure specification and design. Box structures provide a stepwise refinement and verification process based on *black box*, *state box*, and *clear box* forms for defining system behavior and deriving and connecting objects comprising a system architecture.[5,23] Boxes are object-based, and the box structure process provides a systematic means for developing object-based systems.[24] Specifically, the black box form is a specification of required behavior of a system or system part in all circumstances of use, defined in terms of stimuli, responses, and transition rules that map stimulus histories to responses. The state box form is refined from and verified against the black box, and defines encapsulated state data required to satisfy black box behavior. The clear box form is refined from and verified against the state box, and defines procedural design of services on state data to satisfy black box behavior, often introducing new black boxes at the next level of refinement. New black boxes (specifications) are similarly refined into state boxes (state designs) and clear boxes (procedure designs), continuing in this manner until no new black boxes are required. Specification and design steps are interleaved in a seamless, integrated hierarchy affording complete verifiability and traceability.

Box structures isolate and separate the creative definition of behavior, data, and procedures at each level of refinement. They incorporate the essential property of *referential transparency*, such that the information content of an abstraction, for example, a black box, is sufficient to define and verify its refinement into state and clear box forms without reference to other specification parts. Referential transparency is crucial to maintaining intellectual control in complex system developments. Box-structured systems are developed as *usage hierarchies* of boxes, where each box provides services on encapsulated state data, and where its services may be used and reused in many places in the hierarchy as required. Box-structured systems are developed according to the following principles:[25] (1) all data to be defined and retained in a design are encapsulated in boxes, (2) all processing is defined by sequential and concurrent use of boxes, and (3) each use of a box occupies a distinct place in the usage hierarchy of the system. Clear boxes play an important role in the hierarchy by defining and controlling the correct operation of box services at the next level of refinement.

Correctness verification. As noted, in the Cleanroom process, verification of program correctness in team reviews replaces private unit testing and debugging by individuals. Debugging is an inefficient and error-prone process that undermines the mental discipline and concentration that can achieve zero defects. The intellectual control of software development afforded by team verification is a strong incentive for the prohibition against unit testing. "No unit testing" does not, however, mean "no use of the machine." It is essential to use the machine for experimentation, to evaluate algorithms, to benchmark performance, and to understand and document the semantics of interfacing software.

These exploratory activities are entirely consistent with the Cleanroom objective of software that is correct by design.

Elimination of unit testing motivates tremendous determination in developers to ensure that the code they deliver for independent testing is error-free on first execution. But there is a deeper reason to adopt correctness verification—it is more efficient and effective than unit testing. Programs of any size can contain an essentially infinite number of possible execution paths and states, but only a minute fraction of those can be exercised in unit testing. Correctness verification, however, reduces the verification of programs to a finite and complete process.

In more detail, all clear box programs are composed of nested and sequenced control structures, such as sequence, IF-THEN-ELSE, WHILE-DO, and their variants. Each such control structure is a rule for a mathematical function,[9] that is, a mapping from a domain or initial state to a range or final state. The function mapping carried out by each control structure can be documented in the design as an *intended function*. For correctness, each control structure must implement the precise mapping defined by its intended function. The Correctness Theorem[4] shows that verification of sequence, IF-THEN-ELSE, and WHILE-DO structures requires checking exactly one, two, and three *correctness conditions*, respectively. While programs can exhibit an essentially infinite number of execution paths and states, they are composed of a finite number of control structures, and their verification can be carried out in a finite number of steps by checking each correctness condition in team reviews. Furthermore, verification is complete, that is, it deals with all possible program behavior at each level of refinement. The verification process defined by the Correctness Theorem accounts for all possible mappings from the domain to the range of each control structure, not just a handful of mappings exercised by particular unit tests. For these reasons, verification far surpasses unit testing in effectiveness.

Statistical quality certification. In the Cleanroom process, statistical usage testing for certification replaces coverage testing for debugging. Testing is carried out by the certification team based on anticipated usage by customers. *Usage probability distributions* are developed to define system inputs for all aspects of usage, including nominal scenarios as well as error and stress situations. The distributions can be organized into probabilistic state transition matrices or formal grammars. Test cases are generated based on random sampling of usage distributions. The correct output for each test input is specified with reference to an oracle, that is, an independent authority on correctness, typically the software specification. System reliability is predicted based on analysis of test results by a *formal reliability model*, and the development process for each increment is evaluated based on the extent to which the reliability results attained objectives. In effect, statistical usage testing is based on a *formal statistical design*, from which statistical inferences of software quality and reliability can be derived.[3,11,26]

Debugging is an inefficient and error-prone process.

Coverage testing can provide no more than anecdotal evidence of reliability. Thus, if many errors are found, does that that mean that the code is of poor quality and many errors remain, or that most of the errors have been discovered? Conversely, if few errors are found, does that mean that the code is of good quality, or that the testing process is ineffective? Statistical testing provides scientifically valid measures of reliability, such as mean-time-to-failure (MTTF), as a basis for objective management decision-making regarding software and development process quality.

Empirical studies have demonstrated enormous variation in the failure rates of errors in operational use.[8] Correcting high-failure-rate errors has a substantial effect on MTTF, while correcting low-failure-rate errors hardly influences MTTF at all. Because usage-based testing exercises software the way users intend to use it, high-frequency, virulent errors tend to be found early in testing. For this reason, statistical usage testing is more effective at improving software reliability than is coverage testing. Statistical testing also provides new management flexibility to certify software quality for varying conditions of use and stress, by developing special usage probability distributions for such situations. For example, the reliability of infrequently used functions with severe consequences of failure can be independently measured and certified.

Adopting the Cleanroom process

Rigorous and complete Cleanroom implementation permits development of very high quality software with scientific certification of reliability. However, substantial gains in quality and productivity have also occurred in partial Cleanroom implementations.[15,18] Evidence suggests that a phased approach to implementation can produce concrete benefits and afford increased management control. The phased approach, combined with initial Cleanroom use on selected demonstration projects, provides a systematic management process for reducing risk in technology transfer. Three implementation phases can be defined and sequenced in a systematic technology transfer process. The idea is to first introduce fundamental Cleanroom principles and several key technologies in an *introductory implementation*. As team experience and confidence grows, increased precision and rigor can be achieved in a *full implementation* of Cleanroom technology. Finally, an *advanced implementation* can be introduced to optimize the Cleanroom process. Of course, a particular Cleanroom implementation can combine elements from various phases as necessary and appropriate for the project environment.

Introductory implementation. Key aspects of an introductory implementation are summarized in the first row of Table 2. The fundamental idea is to shift from craft-based to engineering-based processes. The development objective shifts from defect correction in unit testing to defect prevention in specification, design, and verification. As experience grows, developers learn they can write software that is right the first time, and a psychological change occurs, from expecting errors to expecting correctness. At the same time, the testing objective shifts from debugging in coverage testing to reliability certification in usage testing. Because Cleanroom code is of high quality at first execution, testers learn that little debugging is required, and they can concentrate on evaluating quality. A management opportunity exists to leverage these technology shifts to develop systems on schedule with substantial improvement in quality and reduction in life-cycle costs.

All development and testing is accomplished by small teams. Team operations provide opportunities for cross-training and a ready forum for discussion, review, and improvement. All work products undergo a team-based peer review to ensure the highest level of quality. The size and number of teams varies according to resource availability, skill levels, and project size and complexity. Teams are organized during project planning and their membership should remain stable throughout development. Cooperative team behavior that leverages individual expertise is a key factor in successful Cleanroom operations.

In any Cleanroom implementation, zero-defect software is an explicit design goal, and measured performance at a target level is an explicit reliability goal. The Cleanroom practices necessary to achieve these objectives require substantial management commitment. Because compromises in process inevitably lead to compromises in quality, it is crucial for managers to understand Cleanroom fundamentals—the philosophy, process, and milestones— and demonstrate unequivocal support. Management commitment is essential to successful introduction of the Cleanroom process.

A key aspect of customer interaction is to shift from a technology-driven to a customer-driven approach, whereby system functional and usage requirements are subject to extensive analysis and review with customers to clearly understand their needs. Maintaining customer involvement in specification and certification helps avoid developing a system that approaches zero defects but provides the wrong functionality for the user.

Unlike the traditional life cycle of sequential phases, the Cleanroom life cycle is based on incremental development. In an introductory implementation, a project is scheduled and managed as a pipeline of increments for development and testing. Functional content and sequencing of increments is typically based on a natural subdivision of system functions and their expected usage. Successive increments should implement user function, execute in the system environment, and accumulate top down into the final product. This incremental strategy supports testing throughout development rather than at completion. It also integrates system increments in multiple steps across the life cycle, to avoid risks of single-step integration of all system components late in a project when little time or resources remain to deal with unforeseen problems.

Management commitment is essential to successful introduction.

In an introductory implementation, a black box specification is written that precisely defines required system functionality in terms of inputs, outputs, and behavior in all possible circumstances of use, including correct and incorrect use. The specification focuses on required system behavior from the user's viewpoint and does not describe implementation details. At this level, specifications are generally expressed in an outer syntax of specification structures, such as tabular formats or variants of Box Description Language (BDL),[5] and an inner syntax of natural language. Cleanroom specifications are important working documents that drive design and certification activities, and they must be kept current for effective team operations. Definition of system user's guides is initiated in parallel with specifications, for elaboration and refinement throughout the development.

In the design process of an introductory implementation, state and clear box concepts are implemented using sound software engineering practices, including stepwise refinement, structured programming, modular design, information hiding, and data abstraction. Successive increments are specified and designed top-down through stepwise refinement, with frequent team review and discussion of design strategies.[8] Stepwise refinement requires substantial look-ahead and analysis, as successive design versions are developed and revised. In this process, a relentless team drive for *design simplification* can result in substantial reductions in the size and complexity of systems, for more efficient correctness verification and subsequent maintenance.

Design with intended functions is a fundamental practice at the introductory level. High-level intended functions originate in system specifications, and are refined into control structures and new intended functions. Expressed primarily in natural language, intended functions are recorded as comments attached to key control structures in designs. Intended functions precisely define required behavior of their control structure refinements. Behavior is defined in functional, nonprocedural descriptions of the derivation of output data from input data. Intended function refinements are expressed in a restricted set of single-entry, single-exit control structures with no side effects, such as sequence, IF-THEN-ELSE, WHILE-DO, and their variants. Each control structure may contain additional intended functions for further refinement. This stepwise specification and design process continues until no further intended functions remain to be elaborated. Intended functions provide a precise road map for designers in refining design structures, and are essential to team verification reviews.

The *last intellectual pass* through a design occurs in team-based correctness verification, another fundamental practice in an introductory implementation. At the design level, verification reviews prove correctness of program control structures, unlike traditional code inspections that trace program flow paths to look for errors. The verification process is based on reading and abstracting the functionality of control structures in designs and comparing the abstractions with specified intended functions to assess correctness. Team members read, discuss, evaluate, and indicate agreement (or not) that designs are correct with respect to their intended behavior. If changes are required, the team must review and verify the modifications before the designs can be considered finished. Verification reviews provide team members with deep understandings of designs and their correctness arguments. Reviews are conducted with the understanding that the entire team is responsible for correctness. Ultimate successes are team successes, and failures are team failures. All specifications and designs are subject to team review, without exception. Fol-

Table 2 A phased implementation for Cleanroom practice

Cleanroom Practice / Implementation	Management and Team Operations	Customer Interaction	Incremental Development	System Specification
Introductory Implementation	• Document an introductory Cleanroom process. • Shift from craft-based to engineering-based processes. • Shift from defect correction in unit testing to defect prevention in specification, design, and verification. • Shift from debugging in coverage testing to quality certification in usage testing. • Shift from individual to small team operations with team review of all work products. • Establish Cleanroom projects and provide commitment, education, and recognition to teams. • Develop to schedule with substantial quality improvement and life cycle cost reduction.	• Shift from technology-driven to customer-driven development. • Analyze and clarify functional requirements with customers to develop functional specifications. • Analyze and clarify usage requirements with customers to develop usage specifications. • Review and validate functional and usage specifications with customers. • Revise functional and usage specifications as necessary for changing requirements.	• Shift from a sequential (waterfall) to an incremental process. • Define increments that implement user function, execute in the system environment, and accumulate top down into the final product. • Define and evolve an incremental development plan for schedules, resources, and increment content. • Carry out scheduled incremental development and testing with stepwise integration of increments.	• Shift from informal, throwaway specifications to precise, working specifications kept current through the project life cycle. • Define specifications of system boundaries, interfaces, and required external behavior in all possible circumstances of use, including correct and incorrect use. • Express specifications in systematic forms such as tables that define required behavior in natural language. • Develop and evolve system user's guides in parallel with specifications.
Full Implementation	• Document a full Cleanroom process. • Increase development rigor with box structure specification, design, and correctness verification. • Increase testing rigor with scientific measures of reliability. • Establish larger Cleanroom projects as teams of small teams with experienced leaders from previous projects. • Develop to schedule with substantial quality and productivity improvement and life cycle cost reduction.	• Educate customers in Cleanroom to increase value, cooperation, and responsiveness to customer needs. • Review black box functional specifications with customers to support increased rigor in specification. • Review usage specifications with customers to support increased rigor in statistical usage testing. • Provide customers with prototypes and accumulating increments for evaluation and feedback.	• Define increments to incorporate early availability of important functions for customer feedback and use. • Rapidly revise incremental plans for new requirements and actual team performance, and respond to schedule and budget changes.	• Develop prototypes as necessary to validate customer requirements and operating environment characteristics. • Define black box specifications in systematic structures such as transition tables expressed in conditional rules and precise natural language.
Advanced Implementation	• Document an advanced Cleanroom process. • Establish a Cleanroom Center of Competency to monitor Cleanroom technology and train and consult with teams. • Establish Cleanroom projects across the organization led by experienced Cleanroom practitioners. • Develop to schedule with substantial quality and productivity improvements and life cycle cost reduction, even in emergency and adverse circumstances.	• Assist customers in leveraging the quality of Cleanroom-developed software for competitive advantage. • Contract with customer for reliability warranties based on certification with agreed usage distributions and reliability models. • Establish cooperative processes with customers for recording operational system usage to calibrate and improve reliability certification.	• Incorporate comprehensive reuse analysis and reliability planning in incremental development plans. • Plan increment content to manage project risk by early development of interface dependencies, critical functions, and performance-sensitive processes.	• Incorporate advances in formal specification methods into local practices. • Develop guidelines for specification formats and conventions based on team experience. • Apply mathematical techniques in black box specifications to define complex behavior with precision. • Express black box specifications where appropriate with specification functions and abstract models. • Develop a specification review protocol for team reviews based on team experience.

	System Design and Implementation	Correctness Verification	Statistical Testing and Reliability Certification	Process Improvement
	• Shift from programming by aggregation of statements to design by stepwise refinement of specifications. • Refine specifications into structured, modular designs using good software engineering practices with substantial look ahead and analysis. • Express designs in control structures and case-structured intended functions expressed in natural language. • Conduct frequent team development reviews to communicate, simplify, and improve evolving designs. • Conduct execution experiments to document the system environment and semantics of interfacing software.	• Shift from unit testing by individuals to correctness verification by teams. • Shift from path tracing in code inspections to functional analysis in verification reviews. • Conduct demonstration verification reviews to set expectations and train teams. • Verify all control structures in team reviews by reading, function abstraction, and comparison to intended functions. • Verify all design changes in team reviews and deliver verified increments to testing for first execution.	• Shift from coverage testing to usage testing. • Define high-level usage distributions in systematic structures such as hierarchical decision trees. • Develop/acquire test cases from a user perspective based on system specifications and usage distributions. • Evaluate quality of each increment through analysis of measures such as failure rates and severity levels. • Return low-quality increments to development for additional design and reverification.	• Shift from informal review of lessons learned to a systematic, documented improvement process. • Measure team productivity, quality, and cost, and analyze for process improvements. • Document improvements to the introductory implementation based on lessons learned from each increment. • Improve or sustain the development process based on quality results of increment testing. • Assess customer satisfaction with Cleanroom-developed systems for process improvements.
	• Refine black boxes (specifications) into state boxes (data designs) and state boxes into clear boxes (procedure designs) and new black boxes. • Define state boxes in data designs and systematic structures such as transition tables expressed in conditional rules and precise natural language. • Define clear boxes in control structures and intended functions expressed in conditional rules and precise natural language. • Encapsulate system data in boxes and define processing by use of box services. • Identify opportunities for reuse of system components.	• Improve introductory practices through increased precision and formality in verification reviews. • Improve verification by introducing mental proofs of correctness based on box structure theory and Correctness Theorem correctness conditions. • Document and reuse proof arguments for recurring design patterns. • Simplify and standardize designs where possible to reduce proof reasoning.	• Establish reliability targets and conduct statistical usage testing for reliability certification. • Define usage probability distributions for all circumstances of use in formal grammers or state transition matrices. • Define alternative distributions for special environments and critical and unusual usage. • Use automated generators to create test cases randomized against usage probability distributions. • Use reliability models to produce statistical reliability measures based on analysis of test results.	• Document improvements to the full implementation based on team decisions in process reviews after each increment. • Use baseline measurements from introductory projects to set quality and productivity objectives. • Improve or sustain the development process based on reliability measurements of each increment. • Conduct causal analysis of failures found in testing and use to identify process areas for improvement. • Conduct surveys of customer satisfaction with Cleanroom-developed systems for process improvement.
	• Incorporate advances in formal design methods into local practices. • Use box structures to document the precise semantics of interfacing software. • Develop guidelines for design formats and conventions based on team experience. • Apply mathematical techniques in state and clear box designs to define complex behavior with precision. • Develop a design review protocol for team development reviews based on team experience. • Establish libraries of reusable, certified designs.	• Incorporate advances in formal verification methods into local practices. • Use trace tables as necessary to support mental proofs of correctness. • Document written proofs of correctness as required for critical system functions. • Develop verification protocols and extended proof rules for application-, language-, and environment-specific semantics.	• Incorporate advances in scientific software certification methods into local practices. • Apply experience of prior Cleanroom projects and customers in setting reliability targets. • Employ usage analysis to validate functional specifications and plan increment content. • Use automated tools to generate self-checking test cases. • Collect customer usage data to track conformance of usage distributions to actual field use. • Apply and evaluate multiple reliability models for best prediction of system reliability in the development environment.	• Use the full rigor of statistical process control to analyze team performance. • Compare team performance with locally-defined process control standards for performance. • Use error classification schemes to improve specific Cleanroom practices in specification, design, verification, and testing.

lowing verification, increments are delivered to the test team for first execution.

In an introductory implementation, usage testing based on external system behavior replaces coverage testing based on design internals. Usage information is collected by analyzing functional specifications and surveying prospective users (where users may be people or other programs). Based on this information, a high-level usage profile is developed, including nominal scenarios of use, as well as error and stress situations. A usage profile can be recorded in systematic structures such as hierarchical decision trees that embody possible usage patterns in compact form. Next, test scenarios are defined based on the usage profile. The idea is that the test cases represent realistic scenarios of user interaction, including both correct and incorrect usage. For example, if particular system functions are used frequently in particular patterns with occasional user mistakes, this usage should be reflected in the test suite. At this stage, the usage profile may not be extremely precise or detailed, but it does contain sufficient information for the test team to generate realistic test cases.

The effectiveness of the development process is measured by system performance in testing with respect to predetermined quality standards, such as failure rates and severity levels. (More precise statistical measures, such as MTTF and improvement ratio, are introduced in the full implementation.) If test results show that the development process is not meeting quality objectives, testing ceases and the code is removed from the machine for redevelopment and reverification by the development team.

Process improvement is a fundamental activity in an introductory implementation. The idea is to shift from informal discussions of lessons learned to a systematic, documented improvement process. Baseline measurements of fundamental project characteristics, such as quality, productivity, and cost, provide a basis for assessing progress and making improvements. The quality results of usage testing can guide changes to the development process. In addition, customer satisfaction with Cleanroom-developed systems can highlight process areas requiring improvements.

Full implementation. Introductory Cleanroom implementation establishes a framework for maturing the process to a full implementation. As summarized in the second row of Table 2, full implementation adds rigor to practices established in the introductory phase through formal methods of box structure specification and design, correctness verification, statistical testing, and reliability certification. For a Cleanroom project of substantial size and complexity, a *team-of-teams* approach can be applied, whereby the hierarchical structure of the system under development forms the basis for organizing, partitioning, and allocating work among a corresponding hierarchy of small teams.

An opportunity exists for more extensive customer interaction in a full Cleanroom implementation. Customers can be provided with education on Cleanroom practices to improve the effectiveness of functional and usage specification analysis and review. In addition, prototypes and accumulating increments can be provided to customers for evaluation and feedback.

Managers and team leaders can leverage Cleanroom experience into additional flexibility in incremental development to accommodate changing requirements, and shortfalls and windfalls in team performance within remaining schedule and budget. Increment planning can emphasize early development of useful system functionality for customer feedback and operational use.

In specification and design, prototyping and experimentation are encouraged to clarify and validate requirements, and to understand and document semantics of interfacing software. The formal syntax and semantics of box structures are used for black, state, and clear box refinements. Black boxes and state boxes are recorded in an outer syntax of formal structures, such as transition tables, with inner syntax expressed in precise *conditional rules*, often given as *conditional concurrent assignments* combined with precise natural language. In clear box design, intended functions are recorded at every level of refinement, expressed in conditional concurrent assignments and precise natural language.

A box-structured system is specified and designed as a hierarchy of boxes, such that appropriate system data are encapsulated in boxes, processing is defined by using box services, and every use of a box service occupies a distinct place in the hierarchy. Box structures promote early identification of common services, that is, reusable objects, that can simplify development and improve productivity. Duplication of effort is avoided when team members have an early awareness of opportunities for use and reuse of common services. Rigorous team verification reviews are conducted for all program structures, using *mental proofs of correctness* based on box structure theory and the correctness conditions of the Correctness Theorem.

Statistical testing involves a more complete and experimentally valid approach than in an introductory implementation. Reliability objectives are established and extensive analysis of anticipated system usage is carried out. Comprehensive specifications of the population of possible system inputs are defined in usage probability distributions recorded in formal grammars or state transition matrices. Automated tools are used to randomly generate test cases from the distributions, and the correct output for each test input is defined based on the system specification. For example, the IBM Cleanroom Certification Assistant (CCA)[27] automates elements of the statistical testing process based on a formal grammar model for usage probability distributions. It contains a Statistical Testcase Generation Facility for compiling distributions (expressed in a Usage Distribution Language) and creating randomized test cases. Reliability models are employed to measure system reliability based on test results, and the development process for each increment is evaluated based on the extent to which reliability results meet objectives. The CCA provides an automated reliability model, the Cleanroom Certification Model, that analyzes test results to compute MTTF, improvement ratio, and other statistical measures. Alternative distributions are often employed to certify the reliability of special aspects of system behavior, for example, infrequently used functions that exhibit high consequences of failure.

Process improvement is established through reviews, following completion of each increment, to incorporate team recommendations into the documented Cleanroom process. Causal analysis of failures and comprehensive customer surveys can provide additional insight into process areas requiring improvement.

Advanced implementation. Key elements of an advanced implementation are summarized in the third row of Table 2. At this level of experience, the Cleanroom process is optimized for the local environment and continually improved through advances in the software engineering technology. A Cleanroom center of competency can be established, staffed by expert practitioners to monitor advances in Cleanroom technology and provide training and consultation to project teams. The Cleanroom process can be scaled up to ever larger projects and applied across an organization. An opportunity exists to achieve Cleanroom quality, productivity, and cost improvements even in emergency and adverse system developments.

Product warranties may be possible in customer contracts, based on certification with usage distributions and reliability models agreed to by both parties. In the future, a capability for developing software with warranted reliability could become a major differentiating characteristic of software development organizations. Customers can benefit by capturing actual usage from specially instrumented versions of Cleanroom-developed systems, to permit test teams to improve the accuracy of usage distributions employed in certification.

Incremental development can be used to manage project risk through early development of key interfaces with pre-existing software, important user functions, and performance-sensitive components. Increments can also be defined to isolate and reduce dependence on areas of incomplete or volatile requirements, and to focus on early initiation of complex, long-lead-time components. Advanced incremental development also includes systematic reuse and reliability planning,[28] facilitated by such tools as the Cleanroom Reliability Manager.[29] In this approach, libraries of reusable components are searched for functions identified in specification and top-level design. If the reliability of candidate components is not known, statistically valid experiments are conducted to estimate reliability. If reliability of a candidate component has previously been certified, the usage profile used in that certification is compared with the new usage profile to determine if the previous certification is valid for the new use. Once reliability estimates exist for new and reused components, an estimate of total system reliability is generated through calculations based on top-level transition probabilities between subsystems. The results of this analysis are used to set reliability requirements for components, evaluate the viability of component reuse, and factor reliability risks into increment planning.

An advanced use of box structure specification involves formal mathematical and computer science models appropriate to the application. Formal black box and state box outer syntax used in full Cleanroom implementation is combined with formal inner syntax expressed as propositional logic, predicate calculus, algebraic function composition, BNF (Backus Naur form) grammars, or other formal notation that affords a clear and concise representation of function. Clear box designs are expressed in design languages for which target language code generators exist, or in restricted subsets of implementation languages, thereby eliminating opportunities for new errors in translation.

In verification reviews, trace tables are employed where appropriate for analysis of correctness, and written proofs are recorded for critical functions, particularly in life-, mission-, and enterprise-critical systems. Application-, language-, and environment-specific proof rules and standards provide a more complete framework for team verification. Locally-defined standards have been shown to be more effective than generic standards in producing consistent practitioner judgment about software quality.[30] In an advanced implementation, the documented process includes environment-specific protocols for specification, design, and verification based on team experience.

In an advanced approach to statistical testing, Markov- or grammar-based automated tools can be used to improve efficiency and effectiveness. For example, the IBM Cleanroom Certification Assistant permits generation of any required number of unique, self-checking test cases. In addition, the rich body of theory, analytical results, and computational algorithms associated with Markov processes have important applications in software development.[31] Both formal grammar and Markov usage models can reveal errors, inconsistencies, ambiguities, and data dependencies in specifications early in development, and serve as test case generators for statistical testing. Initial versions of systems can be instrumented to record their own usage on command, as a baseline for analysis and calibration of usage distributions in certification of subsequent system versions.

An advanced implementation can benefit from a locally-validated reliability model for software certification. Just as locally-validated standards enable more consistent practitioner judgment about software quality, a locally-validated reliability model will enable more accurate prediction of operational reliability from testing results.

In an advanced implementation, the full rigor of statistical process control can be applied to process improvement. Team accomplishments can be compared to locally-defined process control standards for performance. Errors can be categorized according to an error classification scheme to target specific Cleanroom practices for improvement.

Choosing an implementation approach

Cleanroom software engineering represents a shift from a paradigm of traditional, craft-based practices to rigorous, engineering-based practices, specifically as follow.

From:		To:
Individual operations	→	Team operations
Waterfall development	→	Incremental development
Informal specification	→	Black box specification
Informal design	→	Box structure refinement
Defect correction	→	Defect prevention
Individual unit testing	→	Team correctness verification
Path-based inspection	→	Function-based verification
Coverage testing	→	Statistical usage testing
Indeterminate reliability	→	Certified reliability

A phased approach to Cleanroom implementation enables an organization to build confidence and capability through gradual introduction of new practices with corresponding growth in process control. If organizational support and capability is sufficient for full implementation, the highest software quality and reliability afforded by Cleanroom practices can be achieved. Otherwise, a phased implementation is recommended. In general, a software organization that employs informal methods of specification and design, relies on coverage testing and defect correction to achieve quality, and has little experience with team-based operations, can gain the most benefit through an introductory implementation. This first phase introduces a comprehensive set of practices spanning project management, development, and testing, but without the full formality of Cleanroom technology. Once an organization successfully completes a project using the introductory practices, it has prepared itself for a full implementation. Likewise, maturation from full to advanced implementation can occur when the practices of the second stage have been successfully demonstrated.

Note that very few teams in reality will implement the precise set of practices defined within each implementation. Each team embodies unique skills, processes, and experiences that must be assessed when choosing an appropriate implementation. It is often the case that a team can best utilize practices from more than one implementation level. For example, a team using an introductory implementation may have had prior experience with inspections and code reviews. Consequently, it may shift to a full or advanced implementation of the system design and verification practices. Perhaps another mature Cleanroom team, using primarily advanced practices, will find the rigor of the second phase of system specification to be sufficient.

The well-known Software Engineering Institute Capability Maturity Model provides a useful assessment technique to help define the best Cleanroom approach.[32,33] In general, higher assessment levels indicate that an organization can successfully adopt a more complete Cleanroom implementation. Organizations assessed at levels 1 and 2 will likely benefit from an introductory implementation, at levels 2 and 3, a full implementation, and at levels 4 and 5, an advanced implementation.

Phased implementation on the AOEXPERT/MVS project

AOEXPERT/MVS is the largest completed Cleanroom project in IBM, both in terms of lines of code and project staffing. The project adopted an introductory implementation of the Cleanroom process for development, and realized a defect rate of 2.6 errors/KLOC, measured from the first execution of the code. This represents all errors ever found in testing and installation at three field test sites. Development productivity averaged 486 lines of code per person-month, including all development labor expended in specification, design, and testing. In short, the AOEXPERT/MVS team produced a complex systems software product with an extraordinarily low error rate, while maintaining high productivity. The following summary of the project is elaborated in Reference 15.

The AOEXPERT/MVS product. AOEXPERT/MVS is a decision-support facility that uses artificial intelligence (AI) for predicting and preventing complex operating problems in an MVS environment. Primarily a host-based product, it runs in a NetView* environment on MVS with interfaces to several other IBM program products. A workstation component running under Operating System/2* (OS/2*) in the Personal System/2* (PS/2*) environment provides the user interface for the definition and management of the business policies for system operation to be applied by AOEXPERT/MVS to avoid and correct system problems.

Few teams will implement the precise set of practices defined within each implementation.

The complex development environment required expertise in MVS and its subsystems, expert systems technology, real-time tasking, message passing, and windows-based programming for the workstation component. The product was implemented using PL/I, TIRS* (an AI shell), PL/X (an internal IBM system language), assembler, JCL, and REXX for host software, and C and Presentation Manager* for workstation software. The environment was further complicated by two major dependencies on IBM system management products that were developed by other IBM laboratories.

The project began in July 1989, with the first eighteen months spent in the requirements phase. Development team staffing took place during this initial stage. Four departments were ultimately established: one for requirements, two for development, and one for testing. Various support organizations provided market development, quality assurance, information development, usability analysis, and business and legal services.

Table 3 The AOEXPERT/MVS implementation of the Cleanroom process

Cleanroom Practice	Introductory	Full	Advanced
Team operations	X		
Customer interaction	X		
Incremental development		X	
System specification	X		
System design	X		
Correctness verification		X	
Statistical usage testing	X		
Reliability certification	X		
Process improvement	X		

The project team was newly formed, with members ranging from programmer retrainees to senior programmers with 25 years of development experience. The project team averaged 50 people throughout development. Experience in the product domain was mixed, with considerable experience in application development and AI, but very little in MVS and system programming. As it turned out, AI skills were utilized about 10 percent of the time during development, while MVS and system programming skills were needed 90 percent of the time.

This was the first Cleanroom development experience for all participants, with the exception of one development manager and two developers. Consequently, extensive education and training were required to implement Cleanroom practices. The overall project schedule had been established in late 1989, prior to the decision to use the Cleanroom process. Given the schedule and mix of skills and experience levels, the Cleanroom process was first met with healthy skepticism. The team had to grapple with three important factors at once: a new team, little experience in the subject domain, and the new Cleanroom development process.

Defining an introductory implementation. The decision to use the Cleanroom process was made in the second quarter of 1990, a year after the project started and six months prior to the beginning of development. Due to the aggressive project schedule, the large size of the organization, the lack of prior Cleanroom experience, and the limited amount of training time available, the management and technical team decided on a phased implementation of the Cleanroom approach. As summarized in Table 3, the team defined an introductory approach that included team-based operations, external specification of behavior using intended functions, design expressed in a Process Design Language (PDL) with automatic target translation (for PL/I), and staged delivery of each increment to independent testers for first execution. In addition to the introductory practices, two full practices were used: incremental development and team-based correctness verification of every line of code. While it was agreed that statistical testing would be very effective, the test team did not believe it could learn and apply the methodology in time for the first increment. The greatest concern was the late start on defining a usage probability distribution, a task normally initiated as soon as the functional specification is available. The test team initially followed the spirit if not the form of usage testing, with a testing approach based on expected customer usage. Later, statistical usage testing was employed for a significant subset of the product, the workstation component, which accounted for approximately 40 percent of total product code.

Getting started. Cleanroom education was provided to the entire project, with mandatory management participation. To further define the use of Cleanroom process in the project environment, a process working group was formed to document the AOEXPERT/MVS Cleanroom development process, to establish and maintain project procedures, standards, and conventions, to establish and maintain a measurement and improvement subprocess, and to provide a formal mechanism to resolve process issues and make improvements. Each major project functional area, including architecture, host development, workstation development, test, configuration management, and quality assurance, was represented on the process working group. The group documented a comprehensive set of procedures and standards for an integrated, Cleanroom-based software development process. This document and its subsequent use by the team was critical in achieving acceptance and ownership of the process by the team. Changes to the process required approval by the process working group and management. During the development of AOEXPERT/MVS, a number of useful process revisions resulted from suggestions by team members in periodic meetings held to improve the development process.

Applying the introductory implementation. The decision to use the Cleanroom process was made rather late in the project after the product functional specification (PFS) document was almost completed. The PFS is required for IBM program product development, but it is not an adequate replacement for a Cleanroom specification, as it contains only a subset of the information required. The AOEXPERT/MVS team decided to complete the PFS, and then produce a more formal black box, incremental specification. The formal specification used precise English descriptions in conjunction with intended functions to specify the external behavior of the increments.

Following specification, project technical leaders created an incremental development plan that defined the functional content, development schedule, and resource requirements for three software increments. Although the project completion date had been established earlier, substantial flexibility remained for scheduling increment development and testing within the overall schedule of 12 months. Historical productivity and defect rates from comparable traditionally-developed applications were reviewed and the schedules were adjusted based on historical Cleanroom data, personal experience, and confidence. The first increment was planned to contain the least function of the three, in order to quickly familiarize the project team with the new Cleanroom process and development environment. Development of the first increment required two and one-half months, with the second and third increments requiring three and one-half months each.

Eight principal functional components were defined for AOEXPERT/MVS and organized into functional content comprising the three increments. Each component was assigned to a team composed of from one to five developers, with each team augmented by an architect and a tester. Team membership remained stable throughout development of all three increments, helping to ensure continuity and growth of expertise and capability. A functional management approach was adopted because each team consisted of people from different departments. Since each team had a designated team leader, management ownership was assigned based on the team leader. Thus, a manager was responsible for all teams led by members of the manager's department. This process worked well, but required daily communication between managers, usually in the form of morning status meetings where schedules, plans, resources, and performance were addressed.

Following increment planning, development began for the first increment. It immediately became obvious that the developers lacked a good understanding of the entry criteria for team correctness verification reviews. Most understood how to perform verification, but underestimated the level of rigor and precision required in the design material. For example, intended functions documented in many of the early first increment designs precisely specified intended behavior for normal or steady-state operation, but failed to specify intended behavior for error conditions, exception processing, and unexpected input. As a result, the designs could not be verified for correctness.

To address this problem, project management decided that a demonstration verification review of an actual first increment design should be held as early as possible. A senior-level programmer was asked to prepare a design for the review. When the design was ready, his five-member team conducted a formal correctness verification review, with the remainder of the AOEXPERT/MVS organization, numbering about 45 people, in attendance as observers. Everyone in attendance had a copy of the material and followed along with the review team. The review lasted about three hours, with the design failing to pass the verification process. This outcome proved to be an invaluable teaching tool for the project team. Most were surprised that the design did not pass, and even more surprised at the number of changes required to make it verifiable. The demonstration clearly showed the team what was actually expected in a Cleanroom review, and definitely saved a substantial amount of time and frustration in the remainder of the project. Since the first increment was relatively small and straightforward, the team was able to learn how to correctly apply the Cleanroom approach and still make the first delivery date.

Cleanroom facilitators. The AOEXPERT/MVS project benefited from people with prior Cleanroom experience, who played dual roles as team members and Cleanroom methodology consultants. These people served as teachers and advisors, providing guidance on how to write verifiable designs and conduct effective verification reviews. Equally important was the encouragement they gave and confidence they instilled in their peers through their example and coaching. During the first increment of development, one of these ex-

Table 4 AOEXPERT/MVS error rates measured from first execution

AOEXPERT/MVS Project		Industry Expectation		AOEXPERT/MVS Project Results	
Increment	KLOC	Errors at 30/KLOC	Projected Errors	Actual Software Errors	Errors/KLOC
1	16	480	64	43	2.7
2	50	1500	200	41	0.8
3	41	1230	164	97	2.4
Subtotal	107	3210	428	181	1.7
System testing			107	93	0.9
Total	107	3210	535	274	2.6

Where

- Projected errors included increment testing projected at 4 errors/KLOC, and system testing at 1 error/KLOC
- Actual software errors were measured from the first execution
- System testing included system, performance, and field testing

perts was present at every verification review to ensure the methodology was followed, especially with respect to application of the correctness verification conditions. During development of the second and third increments, other team members, now with experience in the Cleanroom process, joined with the original experts to form a core group of five to six facilitators who served a key role in acceptance, application, and improvement of the Cleanroom process.

Team verification reviews. The Cleanroom correctness verification process was closely followed. A check was made prior to every review to ensure that the entry criteria were satisfied, and a disciplined process of correctness condition verification for every control structure was followed during the review process. A moderator was assigned, usually one of the Cleanroom facilitators, to ensure that the reviews were conducted properly, and that all issues were recorded and all changes reverified. The author of the design under verification typically led the team through the review. Also present were a key reviewer, usually the component team leader who had a broad understanding of the component function, and other reviewers, typically members of other teams whose components interfaced with the designs under review. Review materials were required to be distributed to all reviewers at least 48 hours prior to the review, and all reviewers were expected to have read the materials before attending the review.

Quality results. The AOEXPERT/MVS testing process was composed of two phases, increment testing and system testing. (In a full implementation of the Cleanroom process, all testing would be regarded as system testing.) After examining data from prior Cleanroom projects, the test team estimated expected defect rates in testing and customer use of the product. Four errors/KLOC were estimated for increment testing, an additional 1 error/KLOC for system testing, and an additional 0.5 error/KLOC for customer use after the product was shipped. These estimates were significantly lower than those customarily found for comparable products, but the team believed that such aggressive goals should be set, even for a first-time Cleanroom effort.

Table 4 summarizes error rates for the three product increments, measured from the first execution of the code. For comparison, projected errors are shown based on an average industrial rate of 30 errors/KLOC[20] for traditional development projects measured from the first execution of the code, with a total of 3210 errors expected at this rate. The test team estimate of 5 errors/KLOC (4 in increment testing plus 1 in system testing) totaled to 535 errors expected.

The AOEXPERT/MVS team produced the complex systems software product with only 274 errors found in all testing. This error rate of 2.6 errors/KLOC was over an order of magnitude better than the industry average of 30 errors/KLOC, and nearly halved the projected Cleanroom rate of 5 errors/KLOC. A number of system components completed testing with no errors found. For example, five of the eight components in the first 16 KLOC increment proved to be error-free in all testing. In addition, no operational errors whatsoever were found following product installation at three customer test sites, and no post-ship customer errors have been reported to date.

Productivity results. Productivity estimates for AOEXPERT/MVS were based on rates for comparable, traditionally-developed products, modified by expected gains from the Cleanroom process and the belief that productivity would improve with each successive increment. Productivity was estimated at 300 lines of code per person-month (LOC/PM) for the first increment, 350 for the second increment, and 400 for the third increment. Table 5 shows actual productivity rates achieved, based on total lines of code divided by the person-months accumulated for formal specification through testing of the final increment. The person-months include development staff only. The project achieved very competitive productivity rates, exceeding the projected rates by 36 percent overall. This substantial improvement in productivity was a significant factor in enabling the project to meet its schedule. The original code size estimate was 72 KLOC, but the actual code size was significantly larger (107 KLOC) due primarily to unexpected growth in the workstation software (from 10 to 42 KLOC). The growth resulted from the lack of familiarity with OS/2 Presentation Manager and unanticipated requirements. Thus, while actual productivity was a 36 percent improvement over the projected rate, actual code size was 49 percent larger than planned. The increased productivity enabled the team to stay on schedule during the development.

Observations. From the beginning of the project through delivery and testing of the first increment, many developers and testers were somewhat skeptical about the Cleanroom approach. The real turnaround in acceptance occurred after the first increment was delivered and tested and so few errors were found. In fact, several testers were upset and worried when they failed to find any errors; ironically, so were the developers. But this soon changed for everyone—defects quickly became the exception, not the rule, and a "right the first time" psychology took hold.

The challenges facing a new team in an unfamiliar environment were great, and schedules and resources were extremely tight. Nevertheless, a new methodology was introduced, taught, and implemented with substantial success. The primary success factors in this implementation of Cleanroom process were the use of an introductory implementation, early and ongoing management commitment, incremental development of system function, demonstration reviews for team education, team-based peer review of all work products, full application of correctness verification, adherence to defect prevention practices, and the use of Cleanroom consultants and facilitators.

Table 5 AOEXPERT/MVS productivity rates

Increment	KLOC	Projected Productivity LOC/PM	Actual Productivity LOC/PM	% Actual Exceeds Projected
1	16	300	400	+33
2	50	350	500	+43
3	41	400	513	+28
Average		358	486	+36

Where the actual productivity was the LOC/PM measured from formal specification through testing

The AOEXPERT/MVS experience is representative of the new level of quality that is possible in software development today. Cleanroom is a practical and proven alternative to the high cost and poor quality frequently seen in traditional development processes. As evidence of its effectiveness continues to accumulate, the Cleanroom process will be increasingly adopted by organizations seeking competitive business advantage.

Acknowledgments

The growing worldwide community of Cleanroom practitioners and researchers, as well as Cleanroom-developed software owners and users, owes an enormous debt of gratitude to Harlan Mills, a retired IBM Fellow, for his groundbreaking development of theoretical foundations for the Cleanroom process. Mills's insight that programs are rules for mathematical functions became the foundation for a disciplined engineering process of program development that replaced a sea of ad hoc methods. His insight that program reliability can be scientifically certified in formal statistical designs brought the power of statistical quality control to software engineering and management.

Those of us who worked with Mills during the formative years of the Cleanroom process were indeed fortunate to have shared in the exciting intellectual climate he created, and to have contributed to the development of the Cleanroom process.

Acknowledgment is also due to members of the Cleanroom teams whose accomplishments are reported in this paper. These practitioners are setting whole new standards of professional excellence and achievement in software engineering.

The authors would like to thank Mark Pleszkoch of the IBM Cleanroom Software Technology Center for his suggestions.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom Software Engineering," *IEEE Software* **4**, No. 5, 19–24 (September 1987).
2. R. C. Linger, "Cleanroom Software Engineering for Zero-Defect Software," *Proceedings of 15th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA (1993), pp. 2–13.
3. P. A. Curritt, M. Dyer, and H. D. Mills. "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering* **SE-12**, No. 1, 3–11 (January 1986).
4. R. C. Linger, H. D. Mills, and B. J. Witt, "Structured Programming: Theory and Practice," Addison-Wesley Publishing Co., Reading, MA (1979).
5. H. D. Mills, R. C. Linger, and A. R. Hevner, "Principles of Information Systems Analysis and Design," Academic Press, Inc., New York (1986).
6. H. D. Mills, "Certifying the Correctness of Software," *Proceedings of 25th Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Los Alamitos, CA (January 1992), pp. 373–381.
7. M. D. Deck and P. A. Hausler, "Cleanroom Software Engineering: Theory and Practice," *Proceedings of Software Engineering and Knowledge Engineering*: Second International Conference, Skokie, IL, June 21–23, 1990. Knowledge Systems Institute, 3420 Main St., Skokie, IL 60076.
8. E. N. Adams, "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Development* **28**, No. 1, 2–14 (1984).
9. H. D. Mills, "Mathematical Foundations for Structured Programming," *Software Productivity*, Little, Brown and Company, Boston, MA (1983), pp. 115–178.
10. P. A. Hausler, "Software Quality Through IBM's Cleanroom Software Engineering," *Creativity!* (ASD-WMA Edition), IBM, Austin, TX, March 1991.
11. H. D. Mills and J. H. Poore, "Bringing Software Under Statistical Quality Control," *Quality Progress* (November 1988), pp. 52–56.
12. R. C. Linger and R. A. Spangler, "The IBM Cleanroom Software Engineering Technology Transfer Program," *Proceedings of SEI Software Engineering Education Conference*, C. Sledge, Editor, Springer-Verlag, Inc., New York (1992).
13. B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1981).
14. S. E. Green, A. Kouchakdjian, and V. R. Basili, "Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory," *Proceedings of Fourteenth Annual Software Engineering Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD 20771 (November 1989), pp. 1–22.
15. P. A. Hausler, "A Recent Cleanroom Success Story: The Redwing Project," *Proceedings of Seventeenth Annual Software Engineering Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD 20771 (December 1992), pp. 256–285.
16. R. C. Linger and H. D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility," *Proceedings of 12th Annual International Computer Software and Applications Conference (COMPSAC '88)*, IEEE Computer Society Press, Los Alamitos, CA (1988), pp. 10–17.
17. S. E. Green and R. Pajersky, "Cleanroom Process Evolution in the SEL," *Proceedings of 16th Annual Software Engineering Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD 20771 (December 1991), pp. 47–63.
18. L-G. Tann, "OS32 and Cleanroom," *Proceedings of 1st Annual European Industrial Symposium on Cleanroom Software Engineering* (Copenhagen, Denmark), Q-Labs AB, IDEON Research Park, S-223 70 Lund, Sweden (1993), Section 5, pp. 1–40.
19. C. J. Trammell, L. H. Binder, and C. E. Snyder, "The Automated Production Control Documentation System: A Case Study in Cleanroom Software Engineering," *ACM Transactions on Software Engineering and Methodology* **1**, No. 1, 81–84 (January 1992).
20. M. Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, Inc., New York (1992).
21. *A Success Story at Pratt and Whitney: On Track for the Future with IBM's VS COBOL II and COBOL Structuring Facility*, GK20-2326, IBM Corporation (1989); no longer available though IBM branch offices.
22. R. C. Linger and A. R. Hevner, "The Incremental Development Process in Cleanroom Software Engineering," *Proceedings of Workshop on Information Technologies and Systems (WITS-93)*, A. R. Hevner, Editor, College of Business and Management, University of Maryland, College Park, MD (December 4–5, 1993), pp. 162–171.
23. H. D. Mills, R. C. Linger, and A. R. Hevner, "Box Structured Information Systems," *IBM Systems Journal* **26**, No. 4, 395–413 (1987).
24. A. R. Hevner and H. D. Mills, "Box-Structured Methods for Systems Development with Objects," *IBM Systems Journal* **32**, No. 2, 232–251 (1993).
25. H. D. Mills, "Stepwise Refinement and Verification in Box Structured Systems," *IEEE Computer* **21**, No. 6, 23–35 (June 1988).
26. R. H. Cobb and H. D. Mills, "Engineering Software Under Statistical Quality Control," *IEEE Software* **7**, No. 6, 44–54 (November 1990).
27. R. C. Linger, "An Overview of Cleanroom Software Engineering," *Proceedings of 1st Annual European Industrial Symposium on Cleanroom Software Engineering* (Copenhagen, Denmark), Q-Labs AB, IDEON Research Park, S-223 70 Lund, Sweden (1993), Section 7, pp. 1–19.
28. J. H. Poore, H. D. Mills, and D. Mutchler, "Planning and Certifying Software System Reliability," *IEEE Software* **10**, No. 1, 88–99 (January 1993).
29. J. H. Poore, H. D. Mills, S. L. Hopkins, and J. A. Whittaker, *Cleanroom Reliability Manager: A Case Study Using Cleanroom with Box Structures ADL*, Software Engineering Technology Report, IBM STARS CDRL 1940 (May 1990). STARS Asset Reuse Repository, 2611 Cranberry Square, Morgantown, West Virginia 26505.
30. C. J. Trammell and J. H. Poore, "A Group Process for Defining Local Software Quality: Field Applications and Validation Experiments," *Software Practice and Experience* **22**, No. 8, 603–636 (August 1992).
31. J. A. Whittaker and J. H. Poore, "Markov Analysis of Software Specifications," *ACM Transactions on Software Engineering and Methodology* **2**, No. 1, 93–106 (January 1993).

32. M. C. Paulk, W. Curtis, M. B. Chrissis, C. V. Weber, *Capability Maturity Model for Software, Version 1.1*, CMU/SEI-93-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213 (February 1993).
33. M. C. Paulk, C. V. Weber, S. M. Garcia, and M. B. Chrissis, *Key Practices of the Capability Maturity Model, Version 1.1*, CMU/SEI-93-TR-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213 (February 1993).

Accepted for publication October 6, 1993.

Note: At the time of publication, Federal Systems Company, now a unit of Loral Corporation, was an IBM-owned company. Addresses for authors may still be considered valid.

Philip A. Hausler *IBM Corporation, 6710 Rockledge Drive, Bethesda, Maryland 20817 (electronic mail: hausler@vnet.ibm.com).* Mr. Hausler is a senior programmer manager in the Cleanroom Software Technology Center of IBM. His department provides education and consultation for technology transfer of the Cleanroom process. He was a principal developer of IBM's first Cleanroom product, the COBOL Structuring Facility, and has held various development and management positions in IBM. Since 1985, Mr. Hausler has served on the faculty of the Computer Science department at the University of Maryland, Baltimore County, teaching software engineering, programming languages, and compiler theory courses. He received a B.S., *summa cum laude*, in computer science in 1983 from the University of Maryland, Baltimore County, and an M.S. in computer science in 1985 from the University of Maryland, College Park. Mr. Hausler has authored or coauthored numerous refereed papers in technical journals. He is a member of the IEEE.

Richard C. Linger *IBM Corporation, 6710 Rockledge Drive, Bethesda, Maryland 20817 (electronic mail: lingerr@betasvm2.vnet.ibm.com).* Mr. Linger is a member of the Senior Technical Staff of IBM. He is the founder and manager of the IBM Cleanroom Software Technology Center, which is chartered to provide Cleanroom technology transfer services to IBM product laboratories and customers. He worked with Harlan D. Mills in developing the Cleanroom software engineering process, and managed development of the COBOL Structuring Facility product, the first commercial Cleanroom project in IBM. He has written or coauthored two textbooks used in Cleanroom education and over 50 refereed papers on the Cleanroom process, software re-engineering and reverse engineering, and other software engineering topics. Mr. Linger is a member of the ACM and the IEEE.

Carmen J. Trammell *Department of Computer Science, 107 Ayres Hall, University of Tennessee, Knoxville, Tennessee 37996 (electronic mail: trammell@cs.utk.edu).* Dr. Trammell is a research assistant professor and manager of the Software Quality Research Laboratory in the Department of Computer Science at the University of Tennessee. She has held software engineering and management positions in military and commercial product development at Oak Ridge National Laboratory, Martin Marietta Energy Systems, and Software Engineering Technology, Inc. She holds a Ph.D. in psychology and an M.S. in computer science from the University of Tennessee. Dr. Trammell is a member of the ACM and the IEEE.

Reprint Order No. G321-5534.

Cleanroom Process Model

Richard C. Linger
Software Engineering Institute

CLEANROOM PROCESS MODEL

The philosophy behind Cleanroom software engineering is to avoid dependence on costly defect-removal processes by writing code increments right the first time and verifying their correctness before testing. Its process model incorporates the statistical quality certification of code increments as they accumulate into a system.

RICHARD C. LINGER
IBM Cleanroom Software Technology Center

Today's competitive pressures and society's increasing dependence on software have led to a new focus on development processes. The Cleanroom process, which has evolved over the last decade, has demonstrated that it can improve both the productivity of developers who use it and the quality of the software they produce.

Cleanroom software engineering is a team-oriented process that makes development more manageable and predictable because it is done under statistical quality control.

Cleanroom is a modern approach to software development. In traditional, craft-based development, defects are regarded as inevitable and elaborate defect-removal techniques are a part of the development process. In such a process, software proceeds from development to unit testing and debugging, then to function and system testing for more debugging. In the absence of workable alternatives, managers encourage programmers to get code into execution quickly, so debugging can begin. Today, developers recognize that defect removal is an error-prone, inefficient activity that consumes resources better allocated to getting the code right the first time.

This article is based on a paper that appears in *Proc. 15th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 2-13. The *IEEE Software* Editorial Board has selected it as the best practice paper presented at ICSE-15. The board thanks Richard A. DeMillo, ICSE-15 Program Cochair, for his help in arranging for its publication in *IEEE Software*.

Cleanroom teams at IBM and other organizations are achieving remarkable quality results in both new-system development and modifications and extensions to legacy systems. The quality of software produced by Cleanroom development teams is sufficient (often near zero defects) for the software to enter system testing directly for first-ever execution by test teams.

The theoretical foundations of Cleanroom — formal specification and design, correctness verification, and statistical testing — have been reduced to practice and demonstrated in nearly a million lines of code. Some Cleanroom projects are profiled in the box on p. 56.

QUALITY COMPARISON

Quality comparisons between traditional methods and the Cleanroom process are meaningful when measured from first execution. Most traditional development methods begin to measure errors at function testing (or later), omitting errors found in private unit testing. A traditional project experiencing, say, five errors per thousand lines of code (KLOC) in function testing may have encountered 25 or more errors per KLOC when measured from first execution in unit testing.

At entry to unit testing, traditional software typically exhibits 25 to 35 or more errors per KLOC.[1] In contrast, the weighted average of errors found in 17 Cleanroom projects, involving nearly a million lines of code, is 2.3 errors per KLOC. This number represents all errors found in all testing, measured from first-ever execution through test completion — it is the average number of residual errors present after the development team has performed correctness verification.

In addition to this remarkable difference in the number of errors, experience has shown a qualitative difference in the complexity of errors found in Cleanroom versus traditional software. Errors left behind by Cleanroom correctness verification tend not to be complex design or interface errors, but simple mistakes easily found and fixed by statistical testing.

In this article, I describe the Cleanroom development process, from specification and design through correctness verification and statistical usage testing for quality certification.

INCREMENTAL DEVELOPMENT

The Cleanroom process is based on developing and certifying a pipeline of software increments that accumulate into the final system. The increments are developed and certified by small, independent teams, with teams of teams for large projects.

System integration is continual, and functionality grows with the addition of successive increments. In this approach, the harmonious operation of future increments at the next level of refinement is predefined by increments already in execution, thereby minimizing interface and design errors and helping developers maintain intellectual control.

CLEANROOM DEVELOPMENT IS INTENDED TO BE "QUICK AND CLEAN," NOT "QUICK AND DIRTY."

The Cleanroom development process is intended to be "quick and clean," not "quick and dirty." The idea is to quickly develop the right product with high quality for the user, then go on to the next version to incorporate new requirements arising from user experience.

In the Cleanroom process, correctness is built in by the development team through formal specification, design, and verification. Team correctness verification takes the place of unit testing and debugging, and software enters system testing directly, with no execution by the development team. All errors are accounted for from first execution on, with no private debugging permitted.

Figure 1 illustrates the Cleanroom process of incremental development and quality certification. The Cleanroom team first analyzes and clarifies customer requirements, with substantial user interaction and feedback. If requirements are in doubt, the team can develop Cleanroom prototypes to elicit feedback iteratively.

As the figure shows, Cleanroom development involves two cooperating teams and five major activities:

♦ *Specification.* Cleanroom development begins with specification. Together, the development team and the certification team produce two specifications: functional and usage. Large projects may have a separate specification team.

The functional specification defines the required external system behavior in all circumstances of use; the usage specification defines usage scenarios and their probabilities for all possible system usage, both correct and incorrect. The functional specification is the basis for incremental software development. The usage specification is the basis for generating test cases for incremental statistical testing and quality certification. Usage specifications are explained in the section on certification.

♦ *Increment planning.* On the basis of these specifications, the development and certification teams together define an initial plan for developing increments that will accumulate into the final system. For example, a 100 KLOC system might be developed in five increments averaging 20 KLOC each. The time it takes to design and verify increments varies with their size and complexity. Increments that require long lead times may call for parallel development.

♦ *Design and verification.* The development team then carries out a design and correctness verification cycle for each increment. The certification team proceeds in parallel, using the usage specification to generate test cases that reflect the expected use of the accumulating increments.

♦ *Quality certification.* Periodically, the development team integrates a completed increment with prior increments and delivers them to the test team for execution of statistical test cases. The test cases are run against the accumulated increments and the results checked for correctness

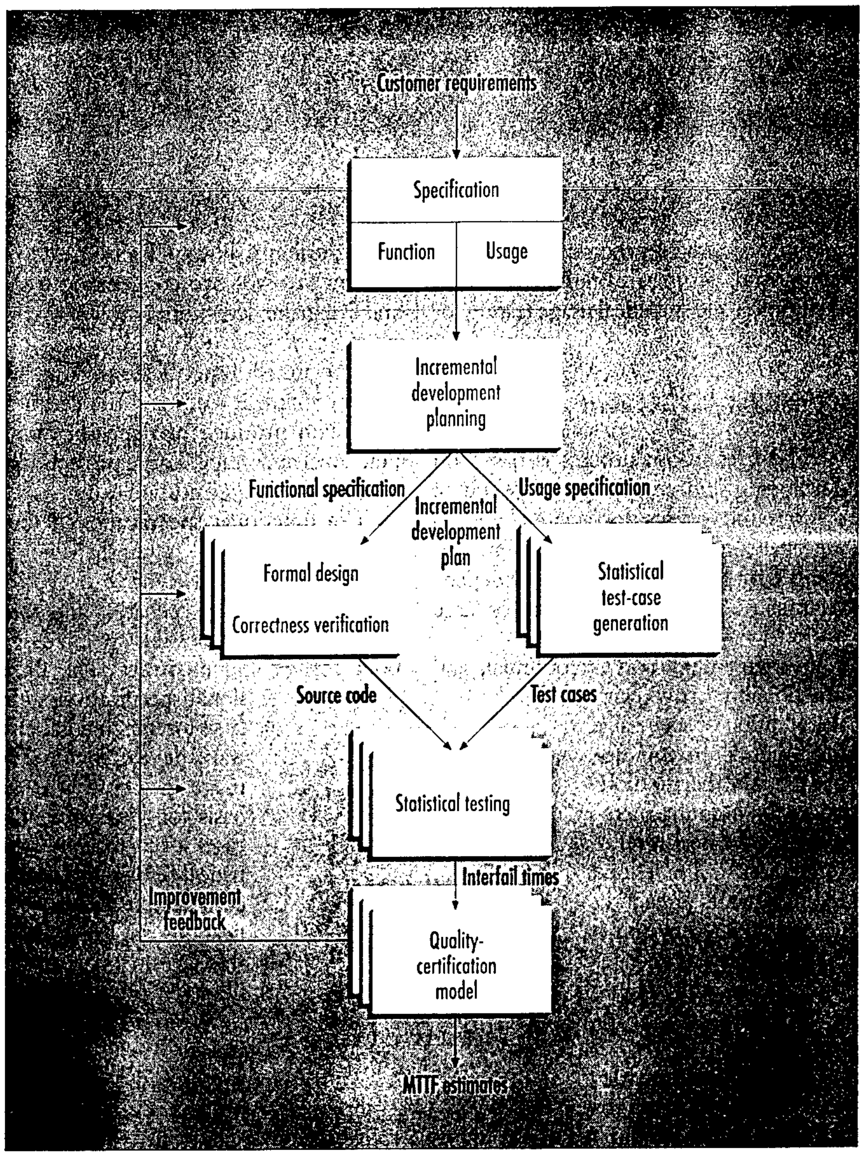


Figure 1. Cleanroom process model. The stacked boxes indicate successive increments.

against the functional specification. Interfail times, that is, the elapsed times between failures, are passed to a quality-certification model[2] that computes objective statistical measures of quality, such as mean time to failure. The quality-certification model employs a reliability growth estimator to derive the statistical measures.

Certification is done continuously, over the life of the project. Higher level increments enter the certification pipeline first. This means major architectural and design decisions are validated in execution before the development team elaborates on them. And because certification is done for all increments as they accumulate, higher level increments are subjected to more testing than lower level increments, which implement localized functions.

♦ *Feedback.* Errors are returned to the development team for correction. If the quality is low, managers and team members initiate process improvement. As with any process, a good deal of iteration and feedback is always present to accommodate problems and solutions.

In the next sections, I describe the specification, design and verification, and quality-certification procedures. A detailed description of increment planning and feedback mechanisms is outside the scope of this article.

FUNCTIONAL SPECIFICATION

The object-based technology of box structures has proved to be an effective technique for functional specification.[3] Through stepwise refinement, objects are defined and refined as different box structures, resulting in a usage hierarchy of objects in which the services of an object may be used and reused in many places and at many levels. Box structures, then, define required system behavior and derive and connect objects comprising a system architecture.[4-5]

In the past, without a rigorous specification technology, there was little incentive to devote much effort to the specification process. Specifications were frequently written in natural language, with inevitable ambiguities and omissions, and often regarded as throwaway stepping stones to code.

Box structures provide an economic incentive for precision. Initial box-structure specifications often reveal gaps and misunderstandings in customer requirements that would ordinarily be discovered later in development at high cost and risk to the project.

They also address the two engineering problems associated with system specification: defining the right function for users and defining the right structure for the specification itself. Box structures address the first problem by precisely defining the current understanding of required functions at each stage of development, so that the functions can be reviewed and modified if necessary. The second problem is critical, especially for large-system development. How can we organize the myriad details of behavior and processing into coherent abstractions humans can understand?

Box structures incorporate the crucial mathematical property of referential transparency — the information content of each box specification is sufficient to define its refinement, without depending on the implementation of any other box. This property lets us organize large-system specifications hierarchically, without sacrificing precision at high levels or detail at low levels.

Box structures. Three principles govern the use of box structures:[4]

♦ All data defined in a design is encapsulated in boxes.

♦ All processing is defined by using boxes sequentially or concurrently.

♦ Each box occupies a distinct place in a system's usage hierarchy.

Each box has three forms — black, state, and clear — which have identical external behavior but whose internals are increasingly detailed.

Black box. An object's black box is a precise specification of external, user-visible behavior in all possible circumstances of its use. The object may be an entire system or any part of a system. Its user may be a person or another object.

A black box accepts a stimulus (S) from a user and produces a response (R). Each response of a black box is determined by its current stimulus history (SH), with a black-box transition function

```
(S, SH) → (R)
```

A given stimulus will produce different responses that are based on history of use, not just on the current stimulus. Imagine a calculator with two stimulus histories

```
Clear 7 1 3
```

and

```
Clear 7 1 3 +
```

If the next stimulus is 6, the first history produces a response of 7136; the second, 6.

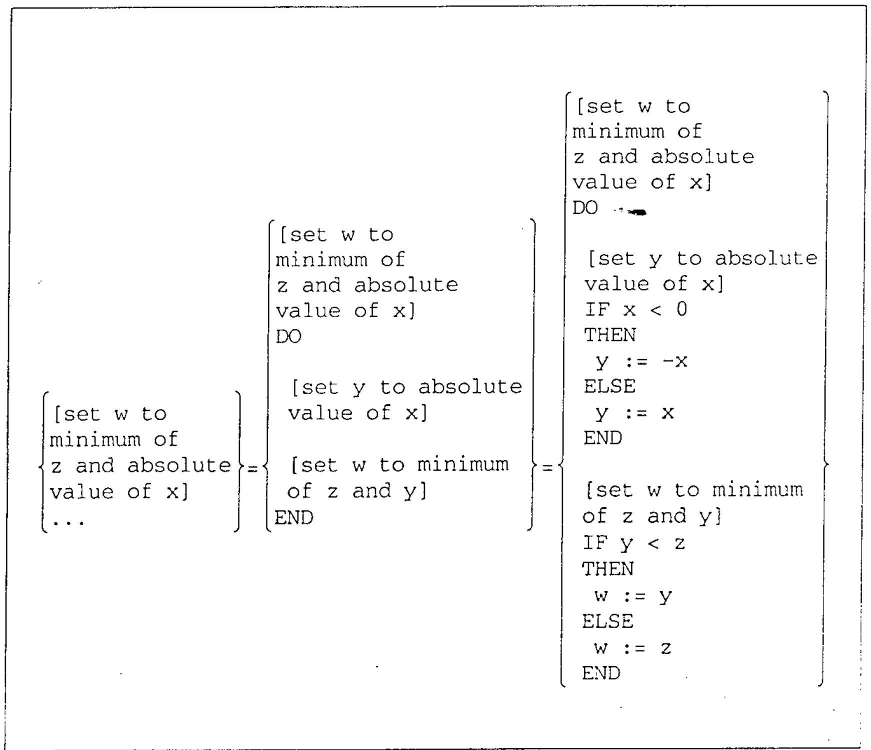
The objective of a black-box specification is to define the responses produced for every possible stimulus and stimulus history, including erroneous and unexpected stimuli. By defining behavior solely in terms of stimulus histories, black-box specifications neither depend on nor prematurely define design internals.

Black-box specifications are often recorded as tables. In each row, the stimulus and the condition on stimulus history are sufficient to define the required response. To record large specifications, classes of behavior are grouped in nested tables and compact specification functions are used to encapsulate conditions on stimulus histories.[6]

State box. An object's state box is derived from its black box by identifying the elements of stimulus history that must be retained as state data between transitions to achieve the required black-box behavior.

The transition function of a state box is

```
(S, OS) → (R, NS),
```



```
[set w to
minimum of
z and absolute
value of x]
...
```

=

```
[set w to
minimum of
z and absolute
value of x]
DO

 [set y to absolute
 value of x]

 [set w to minimum
 of z and y]
END
```

=

```
[set w to
minimum of
z and absolute
value of x]
DO

 [set y to absolute
 value of x]
 IF x < 0
 THEN
  y := -x
 ELSE
  y := x
 END

 [set w to minimum
 of z and y]
 IF y < z
 THEN
  w := y
 ELSE
  w := z
 END
```

Figure 2. Stepwise refinement of a clear-box design fragment that can be verified. Each fragment has identical functional behavior, even though the level of detail increases.

where OS and NS represent old state and new state. Although the external behavior of a state box is identical to its corresponding black box, the stimulus histories are replaced with references to an old state and the generation of a new state, as its transitions require.

As in the traditional view of objects, state boxes encapsulate state data and services (methods) on that data. In this view, stimuli and responses are inputs and outputs, respectively, of specific state-box service invocations that operate on state data.

Clear box. An object's clear box is derived from its state box by defining a procedure to carry out the state-box transition function. The transition function of a clear box is

```
(S, OS) → (R, NS) by procedure
```

So a clear box is simply a program that implements the corresponding state box. A clear box may invoke black boxes at the next level, so the refinement process is recursive, with each clear box possibly introducing opportunities for defining new objects or extensions to existing ones.

Clear boxes play a crucial role in the usage hierarchy by ensuring the harmonious cooperation of objects at the next level of refinement. Objects and their clear-box connections are derived from immediate processing needs at each stage of refinement, not invented a priori, with uncertain connections left to be defined later. The design and verification of clear-box procedures is the focus of the next section.

Because state boxes can be verified with respect to their black boxes and clear boxes with respect to their state boxes, box structures bring correctness verification to object architectures.[4]

DESIGN AND VERIFICATION

The procedural control structures of structured programming used in clear-box design — sequence, alternation (if-then-else), and iteration (while-do) — are single-entry, single-exit structures that cannot produce side effects in control flow. (Control structures for concurrent execution are dealt with in box structures, but are outside the scope of this article.)

When it executes, a given control structure simply transforms data from an input state to an output state. This transformation, known as its *program function*, corresponds to a mathematical function: It defines a mapping from a domain to a range by a particular rule.

For integers w, x, y, and z, for example, the program function of the sequence,

```
              Sequence
Control                Correctness
structure:             condition (for all arguments):

  [f]
  DO
    g;                 Does g followed
    h                  by h do f?
  END

              Alternation
Control                Correctness
structure:             condition:

  [f]
  IF p                 Whenever p is true
  THEN                   does g do f, and
    g                  whenever p is false
  ELSE                   does h do f?
    h
  END

              Iteration
Control                Correctness
structure:             condition:

  [f]                  Is termination
  WHILE p              guaranteed, and
  DO                   whenever p is true
    g                    does g followed
  END                    by f do f, and
                       whenever p is false
                         does doing
                         nothing do f?
```

Figure 3. Correctness conditions in question form for verifying each type of clear-box control structure.

```
DO
  z := abs(y)
  w := max(x, z)
END
```

is, in concurrent assignment form,

```
w, z := max(x, abs(y)), abs(y)
```

For integer $x \geq 0$, the program function of the iteration

```
WHILE
  x > 1
DO
  x := x -2
END
```

is, in English,

```
set odd x to 1, even x to 0
```

Design refinement. In designing clear-box procedures, you define an *intended function*, then refine it into a control structure and new intended functions, as Figure 2 illustrates. Intended functions, enclosed in braces, are recorded in the design and attached to their control-structure refinements. In essence, clear boxes are composed of a finite number of control structures, each of which can be checked for correctness.

Design simplification is an important objective in the stepwise refinement of clear boxes. The goal is to generate compact, straightforward, verifiable designs.

Correctness verification. To verify the correctness of each control structure, you derive its program function — the function it actually computes — and compare it to its intended function, as recorded in the design. A correctness theorem[7] defines how to do this comparison in terms of language- and application-independent *correctness conditions*, which you apply to each control structure.

Figure 3 shows the correctness conditions for the sequence, alternation, and iteration control structures. Verifying a sequence involves function composition and requires checking exactly one condition. Verifying an alternation involves case analysis and requires checking exactly two conditions. Verifying an iteration involves function composition and case analysis in a recursive equation and requires checking exactly three conditions.

Correctness verification has several advantages:

♦ *It reduces verification to a finite process.* As Figure 4 illustrates, the nested, sequenced way that control structures are organized in a clear box naturally defines a hierarchy that reveals the correctness conditions that must be verified. An axiom of replacement[7] lets us substitute intended functions for their control structure refinements in the hierarchy of subproofs. For example, the subproof for the intended function `f1` in Figure 4 requires proving that the composition of operations `g1` and `g2` with intended subfunction `f2` has the same effect on data as `f1`. Note that `f2` substitutes for all the details of its refinement in this proof. This substitution localizes the proof argument to the control structure at hand. In fact, it lets you carry out proofs in any order.

It is impossible to overstate the positive effect that reducing verification to a finite process has on quality. Even though all but the most trivial programs exhibit an essentially infinite number of execution paths, they can be verified in a finite number of steps. For example, the clear box in Figure 5 has exactly 15 correctness conditions that must be verified.

♦ *It lets Cleanroom teams verify every line of design and code.* Teams can carry out the verification through group

```
Program:           Subproofs:

 [f1]              f1 = [DO g1;g2;[f2] END] ?
 DO
  g1
  g2
  [f2]             f2 = [WHILE p1 DO [f3] END] ?
  WHILE
   p1
  DO [f3]
   g3
   [f4]            f3 = [DO g3;[f4];g8 END] ?
   IF
    p2
   THEN [f5]       f4 = [IF p2 THEN [f5] ELSE [f6] END] ?
    g4
    g5
   ELSE [f6]
    g6             f5 = [DO g4;g5 END] ?
    g7
   END
   g8              f6 = [DO g6;g7 END] ?
  END
 END
```

Figure 4. A clear-box procedure and its constituent subproofs. In the figure, each `pi` *is a predicate, each* `gi` *is an operation, and each* `fi` *is an intended function.*

analysis and discussion on the basis of the correctness theorem, and they can produce written proofs when extra confidence in a life- or mission-critical system is required.

♦ *It results in a near-zero defect level.* During a team review, every correctness condition of every control structure is verified in turn. Every team member must agree that each condition is correct, so an error is possible only if every team member incorrectly verifies a condition. The requirement for unanimous agreement based on individual verifications results in software that has few or no defects before first execution.

♦ *It scales up.* Every software system, no matter how large, has top-level, clear-box procedures composed of sequence, alternation, and iteration structures. Each of these typically invokes a large subsystem with thousands of lines of code — and each of those subsystems has its own top-level intended functions and procedures. So the correctness conditions for these high-level control structures are verified in the same way as are those of low-level structures. Verification at high levels may take, and well be worth, more time, but it does not take more theory.

♦ *It produces better code than unit testing.* Unit testing checks only the effects of executing selected test paths out of many possible paths. By basing verification on function theory, the Cleanroom approach can verify every possible effect on all data, because while a program may have many execution paths, it has only one function. Verification is also more efficient than unit testing. Most verification conditions can be checked in a few minutes, but unit tests take substantial time to prepare, execute, and check.

```
[ Q := odd_numbers(Q) || even_numbers(Q) ]
PROC Odd_Before_Even (ALT Q)

   DATA
      odds   : queue of integer [initializes to empty]
      evens  : queue of integer [initializes to empty]
      x      : integer
   END

   [ Q    := empty,                                            wdo
     odds := odds  ||odd_numbers(Q),                            3
     evens := evens ||even_numbers(Q) ]
   WHILE Q <> empty
   DO

      x := end(Q)                                      seq
                                                        1
      [x is odd -> odds := odds || x
       true     -> evens := evens || x ]
      IF odd(x)                                          ite
      THEN                                                2
         end(odds) := x
      ELSE
         end(evens) := x
      END

   END
seq
 1 [ Q    := Q || odds,                   wdo
     odds := empty ]                       3
   WHILE odds <> empty
   DO [end(Q)  := end(odds)]      seq
                                   1
      x       := end(odds)
      end(Q)  := x

   END

   [ Q      := Q || evens,                wdo
   evens:= empty]                          3
   WHILE evens <> empty
   DO [end(Q)  := end(evens)]     seq
                                   1
      x       := end(evens)
      end(Q)  := x

   END

END odd-before-even
```

Figure 5. A clear-box procedure with 15 correctness conditions to be verified. The procedural control structures and the number of correctness conditions that must be checked are shown in bold. Seq indicates a sequence, ite indicates an alternation (if-then-else), and wdo indicates an iteration (while-do).

QUALITY CERTIFICATION

Statistical quality control is used when you have too many items to test all of them exhaustively. Instead, you statistically sample and analyze some items to obtain a scientific assessment of the quality of all items. This technique is widely used in manufacturing, in which items on a production line are sampled, their quality is measured against a presumably perfect design, the sample quality is extrapolated to the entire production line, and flaws in production are corrected if the quality is too low.

In hardware products, the statistics used to establish quality are derived from slight variations in the products' physical properties. But software copies are identical, bit for bit. What statistics can we sample to extrapolate quality?

Usage testing. It turns out that software has a statistical property of great interest to developers and users — its execution behavior. How long, on average, will a software product execute before it fails?

From this notion has evolved the process of *statistical usage testing*,[8] in which you

♦ sample the (essentially infinite) pop-

CLEANROOM QUALITY RESULTS

Cleanroom projects report a *testing error rate per thousand lines of code*, which represents residual errors in the software after correctness verification. The projects briefly described here are among 17 Cleanroom projects, involving nearly a million lines of code, that have reported a *weighted average of 2.3 errors per KLOC found in all testing, measured from first-ever execution of the code* — a remarkable quality achievement.[1]

♦ *IBM Cobol Structuring Facility (Cobol/SF).* This was IBM's first commercial Cleanroom product, developed by a six-person team. This 85 KLOC PL/I program automatically transforms unstructured Cobol programs into functionally equivalent structured form for improved understandability and maintenance. It had a testing error rate of 3.4 errors per KLOC; several major components completed certification with no errors found. In months of intensive beta testing at a major aerospace corporation, all Cobol programs executed identically before and after structuring.

Productivity, including all specification, design, verification, certification, user publications, and management, averaged 740 LOC per person-month. So far, a small fraction of a person-year per year has been required for all maintenance and customer support. Although the product exhibits a complexity level on the order of a Cobol compiler, just seven minor errors were reported in the first three years of field use, all resulting in simple fixes. — R.C. Linger and H.D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM Cobol Structuring Facility," *Proc. Compsac,* IEEE CS Press, Los Alamitos, Calif., 1988, pp. 10-17.

♦ *NASA satellite-control project.* The Coarse/Fine Attitude Determination System (CFADS) of the NASA Attitude Ground Support System (AGSS) was the first Cleanroom project carried out by the Software Engineering Laboratory of the NASA Goddard Space Flight Center. The system, comprising 40 KLOC of Fortran, exhibited a testing error rate of 4.5 errors per KLOC. Productivity was 780 LOC per person-month, an 80 percent improvement over previous SEL averages. Some 60 percent of the programs compiled correctly on the first attempt. — A. Kouchakdjian, S. Green, and V.R. Basili, "Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory," *Proc. 14th Software Eng. Workshop*, NASA Goddard Space Flight Center, Greenbelt, Md., 1989.

♦ *Martin Marietta Automated Documentation System.* A four-person Cleanroom team developed the prototype for this system, a 1,820-line relational database application written in Foxbase. It had a testing error rate of 0.0 errors per KLOC — no compilation errors were found and no failures were encountered in statistical testing and quality certification. The software was certified at target levels of reliability and confidence. Team members attributed error-free compilation and failure-free testing to the rigor of the Cleanroom method. — C.J. Trammell, L.H. Binder, and C.E. Snyder, "The Automated Production Control System: A Case Study in Cleanroom Software Engineering," *ACM Trans. Software Eng. and Methodology*, Jan. 1992, pp. 81-94.

♦ *IBM AOEXPERT/MVS.* A 50-person team developed this complex decision-support facility that uses artificial intelligence to predict and prevent operating problems in an MVS environment. The system, written in PL/I, C, Rexx, and TIRS, totaled 107 KLOC, developed in three increments. It had a testing error rate of 2.6 errors per KLOC. Causal analysis of the first 16-KLOC increment revealed that five of its eight components experienced no errors in testing.

The project reported development team productivity of 486 LOC per person-month. No operational errors have been reported to date from beta test and early user sites. — P.A. Hausler, "A Recent Cleanroom Success Story: The Redwing Project," *Proc. 17th Software Eng. Workshop*, NASA Goddard Space Flight Center, Greenbelt, Md., 1992.

♦ *NASA satellite-control projects.* Two satellite projects, a 20-KLOC attitude-determination subsystem and a 150-KLOC flight-dynamics system, were the second and third Cleanroom projects undertaken at NASA's Software Engineering Laboratory. These systems had a combined testing error rate of 4.2 errors per KLOC. — S.E. Green and Rose Pajerski, "Cleanroom Process Evolution in the SEL," *Proc. 16th Software Eng. Workshop*, NASA Goddard Space Flight Center, Greenbelt, Md., 1991.

♦ *IBM 3090E tape drive.* A five-person team developed the device-controller design and microcode in 86 KLOC of C, including 64 KLOC of function definitions. This embedded software processes multiple real-time I/O data streams to support tape-cartridge operations in a multibus architecture. The box-structure specification for the chip-set semantics revealed several hardware errors. The project had a testing error rate of 1.2 errors per KLOC.

A one-module experiment compared the effectiveness of unit testing and correctness verification. In unit testing, the team took 10 person-days to develop scaffolding code, invent and execute test cases, and check results. They found seven errors. Correctness verification, which required an hour-and-a-half in a team review, found the same seven errors plus three more.

To meet a business need, the third code increment went straight from development, with no testing whatsoever, into customer-evaluation demonstrations using live data. There were no errors of any kind. A total of 490 statistical tests were executed against the final version of the system, with no errors found.

♦ *Ericsson Telecom OS32 operating system.* This 70-person, 18-month project specified, developed, and certified a 350-KLOC operating system for a new family of switching computers. The project had a testing error rate of 1.0 errors per KLOC.

Productivity was reported to have increased by 70 percent for development; 100 percent for testing. The team significantly reduced development time, and the project was honored by Ericsson for its contribution to the company. — L.-G. Tann, "OS32 and Cleanroom," *Proc. 1st European Industrial Symp. Cleanroom Software Eng.*, Q-labs, Lund, Sweden, 1993.

REFERENCES

1. P.A. Hausler, R.C. Linger, and C.J. Trammell, "Adopting Cleanroom Software Engineering with a Phased Approach," *IBM Systems J.*, Mar. 1994, to appear.

ulation of all possible executions (correct and incorrect) by users (people or other programs) according to how frequently you expect the executions to happen,

♦ measure their quality by determining if the executions are correct,

♦ extrapolate the quality of the sample to the population of possible executions, and

♦ identify and correct flaws in the development process if the quality is inadequate.

Statistical usage testing amounts to testing software the way users intend to use it. The entire focus is on external system behavior, not the internals of design and implementation. Cleanroom certification teams have deep knowledge of expected usage, but require no knowledge of design internals. Their role is not to debug-in quality, an impossible task, but to scientifically certify software's quality through statistical testing.

In practice, Cleanroom quality certification proceeds in parallel with development, in three steps.

1. Specify usage-probability distributions. Usage-probability distributions define all possible usage patterns and scenarios, including erroneous and unexpected usage, together with their probabilities of occurrence. They are defined on the basis of the functional specification and other sources of information, including interviews with prospective users and the pattern of use in prior versions.

Figure 6a shows a usage specification for a program with four user stimuli: update (U), delete (D), query (Q), and print (P). A simplified distribution that omits scenarios of use and other details shows projected use probabilities of 32, 14, 46, and 8 percent, respectively. These probabilities are mapped onto an interval of 0 to 99, dividing it into four partitions proportional to the probabilities. Usage-probability distributions for large-scale systems are often recorded in formal grammars or Markov chains for analysis and automatic processing.

In incremental development, you can stratify a usage-probability distribution into subsets that exercise increasing functional content as increments are added, with the full distribution in effect once the final increment is in place. In addition, you can define alternate distributions to certify infrequently used system functions whose failure has important consequences, such as the code for a nuclear-reactor shutdown system.

Program stimuli	Usage-probability distribution	Distribution interval
U (update)	32%	0-31
D (delete)	14%	32-45
Q (query)	46%	46-91
P (print)	8%	92-99

(A)

Test number	Random numbers:	Test cases:
1	29 11 47 52 26 94	U U Q Q U P
2	62 98 39 78 82 65	Q P D Q Q Q
3	83 32 58 41 36 17	Q D Q D D U
4	36 49 96 82 20 77	D Q P Q U Q

(B)

Figure 6. (A) Simplified usage probability distribution for a program with four user stimuli and (B) a sample of associated test cases.

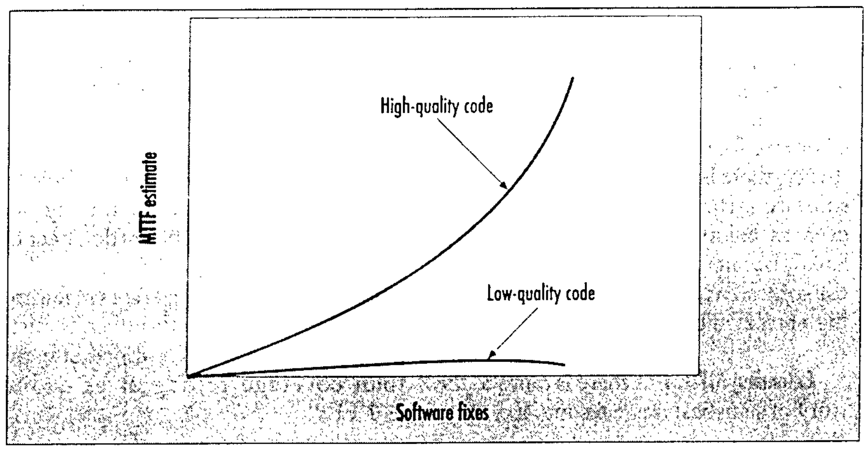


Figure 7. Two sample graphs. The curve for high-quality software shows exponential improvement, such that the MTTF quickly exceeds the total test time. The curve for low-quality software shows little MTTF growth.

2. Derive test cases that are randomly generated from usage-probability distributions. Test cases are derived from the distributions, such that every test represents actual use and will effectively rehearse user experience with the product. Because test cases are completely prescribed by the distributions, producing them is a mechanical, automatable process.

Figure 6b shows test cases for the probability distribution in Figure 6a. If you assume a test case contains six stimuli, then you generate each test by obtaining six two-digit random numbers. These numbers represent the partition in which the corresponding stimuli (U, D, Q, or P) resides. In this way, each test case is faithful to the distribution and represents a possible user execution. For testing large-scale systems, usage grammars or Markov chains can be processed to generate test cases automatically.

3. Execute test cases, assess results, and compute quality measures. At this point, the development team has released verified code to the certification team for first-ever execution. The certification team executes each test case and checks the results against system specifications. The team records execution time up to the point of any failure in appropriate units, for example, CPU time, wall-clock time, or number of transactions.

These *interfail times* represent the quality of the sample of possible user executions. They are passed to a quality certification model that computes the system's quality, including its mean time to failure. The quality-certification model produces graphs like the one in Figure 7.

Because the Cleanroom development process rests on a formal, statistical design, these MTTF measures provide a scientific basis for management action, unlike the anecdotal evidence from coverage testing (If few errors are found, is that good or bad? If many errors are found, is that good or bad?). In theory, there is no way to ever know that a software system has zero defects. However, as failure-free executions accumulate, it becomes possible to conclude that the software is at or near zero defects with high probability.

WE BELIEVE USE OF THE CLEANROOM PROCESS WILL GROW.

Extending MTTF. But there is more to the story of statistical usage testing. Extensive analysis of errors in large-scale software systems reveals a spread in the failure rates of errors of some four orders of magnitude.[9] Virulent, high-rate errors can literally occur every few hours for some users, but low-rate errors may show up only after accumulated decades of use by many users.

High-rate errors are responsible for nearly two-thirds of software failures reported,[10] even though they comprise less than three percent of total errors. Because statistical usage testing amounts to testing software the way users will use it, high-rate errors tend to be found first. Any errors left behind after testing tend to be infrequently encountered by users.

Traditional coverage testing finds errors in random order. Yet finding and fixing low-rate errors has little effect on MTTF and user perception of quality, while finding and fixing errors in failure-rate order has a dramatic effect. Statistical usage testing is far more effective than coverage testing at extending MTTF.[10]

Software that is formally engineered in increments is well-documented and under intellectual control throughout development. The Cleanroom approach provides a framework for managers to plan (and replan) schedules, allocate resources, and systematically accommodate changes in functional content.

Experienced Cleanroom teams can substantially reduce time to market. This is due largely to the precision imposed on development, which helps eliminate rework and dramatically reduces testing time, compared with traditional methods. Furthermore, Cleanroom teams are not held hostage by error correction after release, so they can initiate new development immediately.

The cost of quality is remarkably low in Cleanroom operations, because it minimizes expensive debugging rework and retesting.

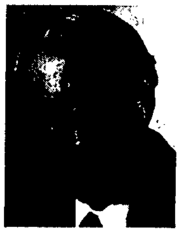
Cleanroom technology builds on existing skills and software-engineering practices. It is readily applied to both new system development and reengineering and extending legacy systems. As the need for higher quality and productivity in software development increases, we believe that use of the Cleanroom process will continue to grow. ♦

ACKNOWLEDGMENTS

I thank Kim Hathaway for her assistance in developing this article. Suggestions by Michael Deck, Philip Hausler, Harlan Mills, Mark Pleszkoch, and Alan Spangler are appreciated. I also thank the members of the Cleanroom teams, whose quality results are reported in this article, and who continue to achieve new levels of quality and productivity.

REFERENCES

1. M. Dyer, *The Cleanroom Approach to Software Quality*, John Wiley & Sons, New York, 1992.
2. P.A. Curritt, M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Trans. on Software Eng.*, Jan. 1986, pp. 3-11.
3. H.D. Mills, R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, San Diego, 1986.
4. H.D. Mills, "Stepwise Refinement and Verification in Box-Structured Systems," *Computer*, June 1988, pp. 23-35.
5. A.R. Hevner and H. D. Mills, "Box Structure Methods for System Development with Objects," *IBM Systems J.*, No. 2, 1993, pp. 232-251.
6. M.G. Pleszkoch et al., "Function-Theoretic Principles of Program Understanding," *Proc. 23rd Hawaii Int'l Conf. System Sciences*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 74-81.
7. R.C. Linger, H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.
8. J.H. Poore and H. D. Mills, "Bringing Software Under Statistical Quality Control," *Quality Progress*, Nov. 1988, pp. 52-55.
9. E.N. Adams, "Optimizing Preventive Service of Software Products," *IBM J. Research and Development*, Jan. 1984, pp. 2-14.
10. R.H. Cobb and H.D. Mills, "Engineering Software Under Statistical Quality Control," *IEEE Software*, Nov. 1990, pp. 44-54.



Richard C. Linger is a member of the senior technical staff at IBM and the founder and manager of the IBM Cleanroom Software Technology Center. His interests are software specification, design, and correctness verification; statistical testing and reliability certification; and the transition from craft-based to engineering-based software development

Linger received a BS in electrical engineering from Duke University. He is a member of the IEEE Computer Society and ACM.

Address questions about this article to Linger at 20221 Darlington Dr., Gaithersburg, MD 20879.

Updated from a presentation made at the
Software Technology Conference (STC), April 1994
and presented at the Second Annual European
Industrial Symposium on Cleanroom Software
Engineering, March 1995

Experience Using CLEANROOM SOFTWARE ENGINEERING in the US Army

S. Wayne Sherer, LCSEC Chief Scientist
Paul G. Arnold, Loral Federal Systems
Ara Kouchakdjian, Software Engineering Technology

Experience Using CLEANROOM SOFTWARE ENGINEERING in the US Army

S. Wayne Sherer, LCSEC Chief Scientist
Paul G. Arnold, Loral Federal Systems
Ara Kouchakdjian, Software Engineering Technology

Abstract

This paper presents the results and lessons learned from a STARS (Software Technology for Adaptable, Reliable Systems) sponsored process technology transfer demonstration. In March of 1992, the Armament, Munitions and Chemical Command (AMCCOM), now the Tank-automotive and Armaments Command (TACOM), Life Cycle Software Engineering Center (LCSEC) at Picatinny Arsenal was selected to demonstrate that Cleanroom Software Engineering (CSE) can be successfully applied in a typical DoD Software Support Activity (SSA). Results indicate that:

- CSE practices can be successfully transferred to a typical DoD SSA,
- engineering staff productivity and product quality were increased while simultaneously increasing job satisfaction, and
- a return on investment of at least 11:1 has been realized on the first project to which CSE techniques were applied.

Introduction

The goal for the technology transfer effort for the LCSEC at Picatinny Arsenal was to conduct a demonstration of CSE practices at a typical DoD SSA.

The LCSEC was selected in response to their expressed interest in improving the process by which they maintain software in general and, specifically, in using the CSE technology. Additionally, as a typical DoD SSA, it was deemed important to improve the processes on which the US Army spends the largest portion of their software money; i.e., in software maintenance and re-engineering (as opposed to new software development). The demonstration was facilitated by Loral and SET (Software Engineering Technology, Inc.).

The LCSEC at Picatinny Arsenal is a representative DoD Software Support Activity that wants to apply a more effective approach to software support. The current state of software re-engineering at the LCSEC varies from project to project but the majority have not achieved the desired level of productivity and quality. A major goal of the Picatinny Arsenal LCSEC is to achieve a Software Engineering Institute Capability Maturity Model (SEI CMM) Level 3 rating by adopting an evolutionary process improvement approach to software re-engineering. Currently, the LCSEC is receiving support, under STARS Task IA02 from Loral and SET, in applying the Cleanroom approach on the re-engineering of the Mortar Ballistic Computer (MBC) into the Improved MBC (I-MBC). Initial results have been successful, the project personnel are employing the Cleanroom engineering practices and adopting the process driven, team oriented approach.

DoD Software Support Activities (SSAs) provide important opportunities to demonstrate STARS efforts to improve software quality and productivity. SSA activities represent a major portion of the DoD software budget and the proportion is expected to be increased during the next decade. This will occur as the many systems in the DoD development pipeline are turned over to SSAs for support. It is likely that, as fewer new systems come into the inventory, DoD managers will attempt to extend the useful life of old systems through software enhancements and re-engineering.

STARS Program

The STARS program is a DoD research and development effort funded under the Advanced Research Projects Agency (ARPA). The main thrust of this effort is that software engineering is process driven, domain-specific, reuse-based, and supported by an integrated software engineering environment. This concept is called Megaprogramming and the STARS program is currently engaged in several demonstration projects of the technologies developed earlier in the program. The Picatinny MBC effort was the first demonstration project to use STARS concepts. Loral is one of the prime contractors for this effort and SET is a principal subcontractor for the Loral/STARS effort.

TACOM LCSEC Overview

The LCSEC at Picatinny Arsenal provides a number of services including: software acquisition support to program managers, computer resource life cycle management plans, pre-planning for software support, manage contracted post deployment software support efforts, software configuration management, and design and implement software changes. The types of battlefield automated systems supported include cannon and tank gun systems; smart mines and munitions; ballistics computers; gunnery simulators; trainers; and nuclear, biological, and chemical detection systems.

The desire for process driven technology was the result of a Software Process Assessment (SPA) conducted by a team of representatives from the AMC LCSECs with coaching from the Software Engineering Institute (SEI). Picatinny LCSEC management has developed a close relationship with the SEI because they desired help with identifying areas to achieve the desired level of productivity and quality. Review of the SPA findings lead LCSEC management to realize that the software engineering process was not under intellectual control. Each new software project, whether performed by contractors or civil servants, was treated largely as new activity that did not necessarily draw on prior experience for process improvement. The only factor that perpetuated experience was people, be it government or contractor, who participated in the same projects time after time. Documentation received by Picatinny, when they were given systems to maintain, was poor or not up to date and no defined process existed for maintaining continual project control. In other words, the state-of-the-practice consisted of traditional software engineering practices that were ad-hoc in nature, as opposed to a disciplined, defined software engineering process. These realizations and the results from the SPA were the basis for their move to enhance their software engineering capabilities.

Typical DoD SSA organizations have immature processes and are subject to morale problems among software engineers due to the combination of an undefined manner of doing work, along with a lack of task-oriented scheduling. The software engineers at the LCSEC did their work well because of individual skills, but often seemed to be stuck in the same "groove," where the same situations, in terms of schedule, would arise year after year. A general lack of enthusiasm pervaded initial discussions with project teams.

Despite these difficulties, however, the customers (various users within the US Army) indicate that they are basically content with the quality of the products. Not many field reports of failures are submitted by their customers, due to extensive, pre-release user testing. Unfortunately, evidence suggests that this may also result from the absence of formal failure observation and reporting mechanisms, making the field quality of developed products difficult to ascertain.

LCSEC management at Picatinny recognized the problems with their state of the practice and took the initiative to recognize Cleanroom Software Engineering (CSE) as the mechanism with which to facilitate the desired cultural, technical and process changes.

Cleanroom Software Engineering (CSE)

CSE was chosen as the process driven technology because it addresses the deficiencies identified during the LCSEC Software Process Assessment (SPA). CSE's management and development team approach was consistent with quality management philosophy, e.g. workforce empowerment, process focus, and quantitative orientation. It provides for the transition of process technology to the project staff and integrates several proven software engineering practices into one methodology. LCSEC management anticipated productivity gains and morale enhancement from the introduction of the technology.

CSE consists of a body of practical and theoretically sound engineering principles applied to the activity of software engineering. Cleanroom consists of a thorough specification phase; resulting in a six part specification, including a precise, **black box** description of the software part of a system. Software development proceeds from the black box specification via a **step-wise refinement** procedure using box-structured design concepts. This process focuses on defect prevention, effectively eliminating costly error removal phases (i.e., debugging) and produces verifiably correct software parts. Development of software proceeds in parallel with a **usage specification** of the software. This usage profile becomes the basis for a **statistical test** of the software, resulting in a scientific certification of the quality of the software part of the system.

A quick high level comparison between the typical development and CSE philosophy of software development is summarized in Table I. The typical development environment can be characterized by craft based techniques which are highly dependent upon the skills of the individuals involved whereas CSE is an engineering discipline with associated rigor and formality.

Table I: Comparison between Typical Development and CSE

Characteristic	Typical Development	CSE
Programs regarded as	Lines of Instructions	Correct rule for a function.
Specification focus	Incomplete description of external behavior and internal design details.	Complete, precise description of external behavior; design details left for development.
Specification to code transformation process	Informal, debugging to verify code.	Stepwise refinement and verification using Box Structures.
Failures are	Expected and accepted.	Unacceptable.
Testing strategy	Futile attempt for coverage and little insight on field reliability.	Random sample based on usage model that predicts field reliability.

Description of Demonstration

To conduct the demonstration, both control and demonstration groups were identified. The control group consisted of a sample set of ongoing and completed software projects at the LCSEC. These projects represent the use of "typical" software engineering methods at the LCSEC. Enhancement projects at Picatinny typically include the correction of observed problems, the addition of new capabilities, and in some cases, re-engineering of software. The demonstration project was the Mortar Ballistics Computer (MBC) re-engineering effort. The demonstration aspect of this project was the adoption of the CSE technology techniques as provided by the participation of Loral and SET. The hypothesis to be confirmed or rejected in this demonstration was: **The use of CSE practices improves the effectiveness (quality and productivity) of the LCSEC software support mission.**

In order to transfer the technology to be demonstrated, as well as the process and culture for a Cleanroom environment, four different tools were employed:

(1) training, in a formal classroom setting which integrated lecture material and numerous hands-on workshops (tailored for this effort),

(2) coaching, both for project planning and execution as well as a medium to promote ongoing education,

(3) process handbooks (evolved for this effort), which act as a written source of educational material and as a reference during project execution, and

(4) an automated process support system (developed for this effort), that helps enforce process adherence and monitors task completion, by automating non creative tasks.

Project Overview

The MBC project was a re-engineering of the current system used by the US Army to aim mortars for combat support. The existing MBC was implemented in DTL (Display Terminal Language) and Z-80 Assembler and is not easily upgraded for new requirements. The re-engineered system is being implemented in Ada and as a result can be moved to new updated hardware platforms. Initial Cleanroom training was provided in November of 1992 and January of 1993, and the MBC re-engineering started in February of 1993. The first two increments of the MBC were completed by January 1994.

In November of 1993, the LCSEC participated in a proposal to take on the responsibility for the Improved MBC (I-MBC) system, which will replace the MBC currently in the field. The other partners in this proposal were other organizations at Picatinny Arsenal. Picatinny had some key win themes that were achieved through the initial STARS cooperation. They had some of the functionality of the proposed I-MBC already developed and certified to be of high quality. They also had shown that by using Cleanroom, they could develop software of high quality with extremely high productivity. This helped support the commitment that the Picatinny Arsenal organization would be successful with the I-MBC project and could do so within acceptable budget limits.

Picatinny won the bid in January of 1994 and started work on the I-MBC in February 1994. The I-MBC will replace the current M23 Mortar Ballistics Computer, and will run on a ruggedized notebook computer. In addition, the I-MBC will add capabilities from other related systems that could not be accommodated by the M23 computer. The I-MBC work began with the development of MBC increment 3 which did not impact the I-MBC effort because the functions performed by the MBC are a subset of those to be performed by the I-MBC. Starting with increment 4, the LCSEC team decided that specification, development and certification for I-MBC were to be created in an incremental manner, similar to the three MBC increments. In order to do this, some risk would be entailed, since it was necessary to make sure the specifications for one increment did not limit options for a future increment. For that reason, the I-MBC team focused on creating a ‘system’ specification, as well as increment specifications. In this manner, risk was minimized, since increment behavior was checked to be consistent with requirements defined in the ‘system’ specification.

The transfer of CSE technology was achieved through formal, classroom-style training courses and follow-on coaching of demonstration team members. The courses involved instruction on the underlying specification, development, and certification methods of CSE and included in-class workshops so that students gained experience applying the technology. As often as possible, workshops were supplemented with examples extracted from the MBC project. Training provided the introduction to and initial experience with the tools that would help enhance individual and team performance.

Project support was given to the team members through repeated on-site coaching visits by CSE experts from Loral and SET. This activity helped to solidify the new ideas as team members saw how the techniques were applied to their specific problems.

The major intent of the training and coaching was to establish the human behavioral changes necessary to develop better software. Implementing CSE is an intellectually challenging process that instills specific values into its participants. For example, the focus on product quality, a major Cleanroom theme, instills a "get it right the first time" attitude into the members of CSE teams. As successes were made and milestones conquered, the CSE teams reported significant improvements in job satisfaction, team spirit, and the desire to continue quality improvements. A significant focus of the coaching effort was to positively reinforce each project success in order to create a stronger identity with the project.

Such behavioral changes within a project are improved by active participation from all levels of the organizational hierarchy from contributing technical leads to engineering management. The initial plan was for the project staffs to work closely as teams, rather than as individuals. Additionally, the intention was for the staffs to be motivated and excited about what they were doing; that is, have a strong identity with the process and project. Thus, coaching contained a "cheer leading" aspect, designed to create a healthy Cleanroom environment.

Reinforcement of CSE was provided through the availability of a six volume set of process manuals to the demonstration groups. These process manuals were an integral part of the training program and were discussed in detail, both during the formal training sessions and off-line as a part of the follow-on coaching activities. The process manuals augment the training by providing reference information to LCSEC engineers, using Cleanroom concepts, and serve as a reference source for resolving questions about specific issues concerning process adherence. The process manuals are organized as follows:

Volume 1:	Cleanroom Engineering Process Introduction and Overview
Volume 2:	Organization and Project Formation in the Cleanroom Environment
Volume 3:	Project Execution in the Cleanroom Environment
Volume 4:	Specification Team Practices
Volume 5:	Development Team Practices
Volume 6:	Certification Team Practices

The division of the volumes represents a separation of concerns for the various project stakeholders. Each volume defines the tasks and the control flow between the tasks necessary to conduct the specific process which is the focus of the manual. Engineering processes are defined as formal control-flow procedures with specific completion conditions. Collections of engineering processes also have the same level of formalized control flow and completion conditions. Thus, each engineer, manager or other staff member has well defined roles and tasks that exist as a part of a larger software process.

CSE is a formal process that clearly defines the tasks necessary for the engineering effort to progress, the completion conditions for each task, and the control flow that dictates the order of work on each task. Process management entails the use of a clearly defined process as the approach to be used to complete the particular project. The intent with process management is to give engineers a clear and understandable road map which they can follow and by which they may track progress towards project completion.

Awareness of software process is a key issue in successfully transferring technology to an organization and to an organization's long term success with applying CSE. The project staffs at the Picatinny LCSEC received an introduction to process definition and process management in the context of CSE. Coaching also reinforced the importance of following the defined process and using the process definition, which defines the possible project alternatives, to support the selection of correct project choices.

Baseline Metrics

The control groups represent the state-of-the-practice at the LCSEC. Baseline metrics were collected in order to gain insight into project practices and to establish a basis of comparison to the demonstration Cleanroom groups. The Control Group Projects consisted of software engineering efforts in Fortran, Z-80 Assembler, Motorola 68000 Assembler and DTL (Display Terminal Language). A LOC is defined as a carriage return for these baseline control group projects. This definition was used because of the wide difference in software development languages contained within the control group projects. Table II presents the baseline metrics for the control group. These metrics are presented with the caution that some data collection mechanisms are unreliable, resulting in inaccuracies. The numbers in Table II are similar to results reported by Mosemann for other projects within the DoD [Ada and C++: A Business Case Analysis, July 1991].

Table II: Baseline Metrics for Control Group Projects

Project / Measure	Control Group Projects
Number of Projects	5
Range of Effort - Staff Months	21-58
Total Technical Staff Months	192
Total KLOC (*)	23.14
DERIVED METRIC: Productivity - LOC/Staff Month	121

Observations

The following observations are a compilation of experiences with the MBC and I-MBC teams. These observations are in the context of Loral's and SET's other experiences with replacing craft-based practices with engineering-based practices, both in the private sector and with government organizations. One must keep in mind these observations are preliminary since the project has not been completed.

1. *The assigned project teams were able to assimilate and even adapt the Cleanroom Software Engineering practices.*

A common worry among managers when hearing about Cleanroom is that it is too hard or too mathematical for their staff. At Picatinny, engineers were able to apply and adapt the Cleanroom practices to the needs of their project. Engineers learned, used and extended the ideas successfully for their project. The evidence of this observation is the products they have produced.

Disciplined engineering in a team environment requires rigor, cooperation of individuals, and the creativity to apply theory to real world problems. This creates a challenging work environment that tends to bring out the best in both individuals and teams.

A prime example of the accomplishments of the MBC team was the tailoring of the box structures algorithm to meet both their application environment and the target programming language, Ada. MBC team members have made original contributions to the expression of box structure constructs in Ada, which will have applicability across many Cleanroom projects. This has benefited both the project, in terms of constructive methods, and the individual team members, in terms of a sense of accomplishment. Building on this accomplishment, additional improvements for documenting the black box were later determined. The development team felt that the initial specification approach used for the first three increments, was insufficient, in that the information provided was incomplete and not easily readable. Through a great deal of interaction between the specification and development teams, the specification team adopted a documentation approach that fulfilled the development team's requirements.

The team has enjoyed using the various Cleanroom techniques and have seen many real accomplishments. The specification team is convinced that this is the most complete and precise specification they have ever seen. The step-wise refinement and verification, which drives engineers to define one small step to take at a time, take that step, and then confirm its correctness, has also been successful. The development team is convinced that they have a great design and have minimized the amount of code they need to develop.

2. A *process driven approach supports engineers in mastering a new technology.*

Process driven project management is one of the two basic technologies being advanced by the STARS program. The Picatinny project was the first project on which this key idea has been employed.

The reason process driven management seems to support technology transfer can be summarized as follows. When doing something for the first time, one often asks, "What do I do next?" or "When will I be done?" This indicates a lack of understanding the big picture, where engineers can clearly place their efforts in a project context. This is not only an attribute of first time usage of techniques or a process, but also an indication that a clearly defined process does not exist or is not effectively managed.

By placing the Cleanroom techniques within a fully defined process, LCSEC engineers knew precisely what step they were currently on, as well as what had been completed and what remained to be done. Giving each individual the foresight that showed where they were in the context of the entire project strengthened project identity and boosted morale.

3. *Staff morale has improved on the project teams.*

Another common fear of managers when hearing about Cleanroom is that their staff members will not like it due to the rigor of the process and the absence of "positive" feedback through executing their own code. This has not the been the case at other places where cleanroom has been introduced, and Picatinny is no exception. When an organization replaces craft-based practices with engineering-based practices, morale improves. One reason seems to be that people now know what to do, when to do it, and how it should be done. This eliminates the uncertainty and anxiety that results when one has to not only do a job but also has to determine what to do and when. Additionally, when engineers learn to use the Cleanroom practices, they know they can do the high quality job they have been striving to achieve. Engineers are convinced that they are producing a better product. As a result, they are excited about it.

At the Picatinny LCSEC, all the engineers, both in informal contacts and in a questionnaire distributed to the engineering staff, reported morale improvements. The LCSEC management has also confirmed the existence of the improved morale and, of course, is favorably impressed.

4. *Facilitation of work effort is greatly enhanced through process driven management.*

Team leaders managed by process definition and task lists which allowed more visibility of project status by management. The intent with process management is to give engineers a clear and understandable road map which they can follow and by which they may track progress towards project completion. Awareness of software process is a key issue in successfully transferring technology to an organization and to an organization's long term success with applying a given process driven approach to project management.

5. *The team-oriented approach of CSE saw immediate acceptance and realized both tangible and intangible benefits.*

A key ingredient of Cleanroom is that a team amplifies human performance. People took advantage of the insight of others in order to bring about the best possible project result. Good people working together produce better results. The simple idea that many minds are better than one makes the outlook for quality good. However, some less tangible benefits were realized as well. The fact that the entire team is responsible for quality, in a series of checks and reviews, puts pressure on the team and **not** on individuals. This pressure creates a reliance on team activity over individual performance. Furthermore, as successes are encountered, the entire team takes credit, not a single individual, thus, cementing the teamwork concepts. The bottom line is that teamwork improves individual performance.

Our observation is that the I-MBC team now works within an effective team-oriented environment. We believe that further use of Cleanroom will establish a strong team mentality that will serve to further improve the initial good results.

6. *Coaching is a key ingredient of technology transfer success.*

Although the training was rigorous with a mixture of theory and workshops, students learn at different rates. Coaching allowed Loral and SET staff to re-educate the slower-to-adopt project staff members and keep the entire team on a common level of knowledge and expertise. Loral and SET technical presence at project inception and during project execution helped solidify the transfer of the technology and ensured that the project got started in the most efficient manner.

Furthermore, there was a gap between the end of training and the start of the project and some of the education was forgotten. Coaching became the mechanism to re-educate and supplement the original training. Further, as good ideas were conceived by some team members, it was possible to see that all members were supplied with the new ideas.

As the project progressed, the CSE ideas needed to be adapted to the specific Picatinny environment. Coaches were used to discuss design alternatives and to help in refining the technology to best serve the application.

Perhaps the most unnoticed but effective use of coaching was in the positive reinforcement the CSE coaches were able to give to the team members and the team as a whole. Coaches are recognized as experts. When experts comment positively on original ideas by a team member, the effect can be enormous in terms of self-esteem and sense of accomplishment and contribution. The CSE trainers tried to positively reinforce the behavior of those making such contributions and encourage others to seek answers beyond the limits of current knowledge. The "cheer leading" approach increased project satisfaction, which motivated greater project performance. Based on this experience, it is now believed that coaching should be a formal part of any technology transfer effort.

7. *Communication among teams (and between team members) is greatly enhanced through a process driven CSE approach.*

An important ingredient of any process driven activity is communication among contributing teams and individuals. One aspect of this was that no team culture existed at the LCSEC; meaning that no real notion existed of how teams are *supposed* to behave during project execution. This problem manifested itself in many different ways. Testers often did not receive specification updates (and failed to ask for them). Also, work tended to be duplicated by multiple personnel because the division of tasks was unclear and communication among members occurred too seldom.

Two approaches were used to solve this problem at Picatinny. The first was to establish effective communication among team members and the second was to establish communication among the different departments involved in the project. The adoption of a well defined process includes a vocabulary that is of great help to the understanding and discussion of the process. This well defined vocabulary makes communication between team members much more effective and productive. The improved communication also started a shift in the culture of the teams. Team members report that they readily use each other as information sources, quality checks, etc. Team reviews are effective and informative. However, the second aspect, communication between departments, continues to be a problem. The MBC certification team members work for a different department than the specification and development teams. Resulting problems are that the certification team finds themselves working from outdated specifications. Furthermore, the certification team seems to duplicate each other's work. A future goal is to be able to duplicate the success of the specification and development teams in the certification team, primarily by improving communications. A more concerted effort should have been made by the coaches to minimize these communications problems.

8. *Specific techniques of Cleanroom Software Engineering were easily and successfully used.*

Three specific techniques were identified by project staff as being major sources of their improved performance. These techniques are team reviews, Cleanroom specifications and box structured design, and are described in greater detail below:

Team reviews, although slow and awkward at the start, were cited by team members as one of the most successful aspects of CSE. Members report that "team shared responsibility" eased misgivings about participating in such a big project. This negated "finger pointing" that existed in previous projects and allowed even difficult personality combinations to work together. The result was that everyone participated and worked as a team toward project success and completion. Morale increased sharply as groups of individuals transformed into an effective software team.

Cleanroom specification, most notably black box documentation, was cited as being responsible for gains in productivity. Many talented engineers existed on the project and their productivity was significantly enhanced when working from a well defined problem statement. The completeness of the specification was the main reason cited for the team's confidence that they were producing a high quality product.

Box structured design is credited with focusing the code generation process and with making team reviews more effective. The team enjoyed the orderly process of developing software. It got them started more quickly on solving a particular problem and they were able to measure the progress of the development activity with more precision than in the past. Since the process relies a great deal on logical thinking as opposed to programming skill, less experienced programmers are able to take a bigger share of the development burden. Therefore, software engineers can make the most of their software engineering skills without having to develop in-depth programming language expertise.

Results

The most important result noted by this effort, even in its preliminary form, is that the motivation to continue to use Cleanroom practices at Picatinny has been established. This demonstration effort was sponsored by STARS and the continued effort is being sponsored by the TACOM LCSEC. This result is an instance of the STARS program fulfilling its mission by being the catalyst for introducing improvements to the software engineering capabilities in the DoD. The decision to expand CSE usage across the entire organization, is the most definitive conclusion of this effort.

In addition to the above mentioned conclusion, the following conclusions can be drawn based on the current status of the MBC/I-MBC project.

1. *It is possible to transfer CSE practices to project teams operating within a typical immature DoD SSA organization.*

This was shown by the fact that the MBC and I-MBC project has progressed to a point where CSE is being successfully applied. This result shows that a specific CMM maturity rating is not necessary in order to benefit from Cleanroom Software Engineering. The engineering staff also enjoyed using the ideas, and all were interested in using the ideas again. Additionally, nearly all were interested in supporting and participating in the establishment of the planned "Cleanroom Competency Center" at the Picatinny Arsenal.

2. *Typical DoD SSA organizations can realize important benefits, in terms of improved process productivity, product quality, and staff morale, from the application of CSE.*

This conclusion is supported by the tripling of productivity of the MBC team. The LOC were calculated the same way for this Ada Language based code as was done for the baseline control group projects with the exception that blank lines of code were not counted. Table III shows the results for productivity for increments 1 through 4. Table IV shows the raw numbers that were used to calculate the productivity shown in Table III. Future increments should show additional improvement.

Table III: Productivity Change for MBC/I-MBC

Increment	**LOC per Staff Month**	**Change in Productivity based on Picatinny Baseline Metrics**
1	370	3.1 : 1
1 + 2	519	4.3 : 1
1 + 2 + 3	see note	see note
1 + 2 + 3 + 4	443	3.7 : 1

Note: Staff months data lost due to accounting mistake but included in next entry

$$\text{Change in Productivity} = \frac{\text{LOC per Staff Month}}{\text{121 LOC Baseline Productivity}}$$

Table IV: Cumulative Raw Numbers for MBC/I-MBC

Increment	**MBC Staff Months**	**Coaching Staff Months**	**Training Staff Months**	**Lines of Code (LOC)**
1	23	3	5	8,500
1 + 2	42.5	6	5	22,063
1 - 3	see note Table III	8.5	5.5	28,579
1 - 4	86.1	10	6	38,155

Early indications are that quality appears to have improved over previous product quality according to Picatinny's customers. The quality of the software (See Table V) developed is very high when compared to quality for traditional software development. Thus, the result achieved will be viewed by the team as the mark to better on the next increment of this project. The incentive and motivation for continual improvement is firmly in place among the team members.

Table V: Software Quality

Failure Type	**Increment Failures**				**Description**
	1	**2**	**3**	**4**	
Process	19	6	0	0	Approved improvements made to design but not reflected in updates to the specifications for certification.
Spelling	3	0	1	0	Misspellings on displays.
Behavior	2	3	3	6	Coding Errors, did not work as specified.
Total	24	9	4	6	Total numbers of failures
Quality	2.82	.41	.14	.16	Failures/KLOC

The MBC project has realized significant gains from the CSE ideas. Once the learning curve had been completed, initial successes in creating the Black Box specification served to cement commitment to CSE.

The resulting conclusions from the overall evaluation are preliminary because the demonstration project is still in its early stages. However, the original hypothesis that Cleanroom improves the effectiveness of the software re-engineering activities at Picatinny looks very promising. Indeed, management and staff agree that morale and motivation is extremely high, that teamwork is now the normal mode of operation, and that people are excited about the software process being established and are motivated to produce high quality products.

A good technical road map is in place at Picatinny; the technical personnel are developing the skills that appear to show significant gains in productivity. Even more promising is the fact that these gains were made on the first use of CSE

3. *The return on investment at Picatinny cannot be definitively calculated, but indications are that there is a significant return on investment.*

Since the project is not yet complete, a preliminary estimate of return on investment can only be based on estimates from the information currently available. The resulting return on investment (ROI) calculations appear below in Table VI.

Table VI: Return on Investment for MBC/I-MBC

Increment	LOC per Staff Month	Staff Months MBC would have taken without CSE	ROI Base	ROI with Coaching
1	370	8500 LOC/121 LOC/SM = 70.3	5.9:1	8.9:1
1 + 2	519	22063 LOC/121 LOC/SM = 182.3	12.7:1	26.7:1
1 - 4	443	38155 LOC/121 LOC/SM = 315.3	14.3:1	36.5:1

SM = Staff Months

$$\text{ROI Base} = \frac{\text{SM MBC would have taken - SM MBC took}}{\text{Staff Months of Coaching Effort + Staff Months spent in Training}}$$

$$\text{ROI with Coaching} = \frac{\text{SM MBC would have taken - (SM MBC took + SM of Coaching Effort)}}{\text{Staff Months spent in Training}}$$

4. *An Automated Process Support System (PSS), that is consistent with the process defined for the project, facilitates technology transfer*

Automating the non-creative tasks of a new technology, such as file access and simple process flow facilitates the adoption of the new technology.

5. *Based on this demonstration we now believe that a technology transfer program to support individual projects at a typical DoD SSA organization must begin with a defined process for the project and should consist of the following components:*
(1) a technology transfer plan,
(2) formal CSE training ,
(3) the availability of qualified coaching,
(4) the availability of engineering handbooks.

The combination of technology transfer components created a series of successes at Picatinny; including productivity gains, expected quality gains, and the increased motivation of the engineering staff.

Future Directions

The next steps for LCSEC/STARS cooperation have been defined. The impressive results to date in the areas of productivity, quality, return on investment and moral have convinced TACOM LSCEC management to continue the work begun under this demonstration project and to expand it further throughout the organization and this includes the following:

Creating a Cleanroom Competency Team to build CSE and process expertise in house. This group will be responsible for continuous review and study of the application of CSE at Picatinny.

Evolve Cleanroom Software Engineering into a complete life cycle process as far as is feasible using standard Cleanroom techniques. Work is currently underway to perform a mapping of the evolved Cleanroom Software Engineering process, as used at Picatinny, against SEI developed Software Process Frameworks (SPF) for the SEI CMM Levels 2 and 3. The results of this mapping will provide Picatinny with a road map of all areas that are either not covered or are poorly covered by the currently documented CSE process. These results will be used to identify areas for Cleanroom process definition extensions. Areas that are not practical for Cleanroom extension will be covered with non Cleanroom processes. The result will be a complete life cycle process definition for Picatinny to support their move to SEI CMM level 3.

Why Isn't Cleanroom the Universal Software Development Methodology?

Johnnie Henderson
Loral Space Information Systems

Why Isn't Cleanroom the Universal Software Development Methodology?

Johnnie Henderson, Loral Space Information Systems

Cleanroom - a methodology that promises much lower error rates, higher productivity, delivery of software on schedule and within budget - sounds like the proverbial "silver bullet" that the industry is looking for. Why hasn't it caught on and spread like wildfire? There are three basic reasons: a belief that the Cleanroom methodology is too theoretical, too mathematical and too radical for use in real software development, it advocates no unit testing by developers but instead replaces it with correctness verification and statistical quality control-- concepts that are directly opposite of how most software is developed today, and maturity of the software development industry. Use of Cleanroom processes requires rigorous application of defined processes in all lifecycle phases. Since most of the industry is still operating at the ad hoc level (as defined by the SEI CMM), the industry has not been ready to apply those techniques.

What does the experience in using Cleanroom say about whether or not these are valid concerns? Following is a brief discussion of those concerns:

Correctness verification is too theoretical to be usable in real software development. The fundamental approach to verification as espoused by Cleanroom is aimed at introducing mathematical reasoning, not mathematical notation into the verification process. The principal motivation is to provide a rigorous methodology for software development and to provide a firm foundation as an engineering discipline. Mathematical verification of programs is done by using a few basic control structures and defining proofs following rules specified in a correctness theorem. The proof strategy is divided into small parts that easily accumulate into proof for a large software system[1].

The method of human mathematical verification used in Cleanroom is called functional verification. Functional verification is organized around correctness proofs, which are defined for the design constructs used in a software design. Using this type of functional verification, the verification problem changes from one with an infinite number of combinations to consider to a finite process because the correctness theorem defines the required number of conditions that must be verified for each design construct used. It reduces software verification to ordinary mathematical reasoning about sets and functions[2]. The objective is to develop designs in concert with associated correctness proofs. Designs are created with the objective of being easy to verify. A rule of thumb followed is that when designs become difficult to verify, they should be redone for simplicity[1,2].

The Cleanroom methodology has been used to develop a variety of types of applications, most of the applications in software that has been sold commercially or embedded in operational application systems[3]. Individuals involved in these development efforts were a cross section of the software engineering population and were able to apply the Cleanroom method after a brief but intensive training program[4].

Unit testing vs. statistical quality control. Statistical quality control is used when you have too many items to test all of them exhaustively. Instead, you statistically sample and analyze some items and scientifically assess the quality of all of the items through extrapolation. This technique is widely used in manufacturing in which items in a production line are sampled, the quality is measured, then sample quality is extrapolated to the entire production line, and flaws are corrected if the quality is not as expected.

For software, this notion has been evolved so that you perform statistical usage testing--testing the software the way the users intend to use it. This is accomplished by defining usage probability distributions that identify usage patterns and scenarios with their probability of occurrence. Tests are derived that are generated based on the usage probability distributions. System reliability is predicted based on analysis of the test results using a formal reliability model, such as mean-time-to-failure[4].

The underlying concern is that random, statistical-based testing will not provide sufficient coverage to ensure a reliable product is delivered to the customer. The coverage concern stems from a misapprehension that statistical implies haphazard, large, and costly and that critical software requirements, which may be statistically insignificant, are overlooked or untested. Coverage is directly related to the robustness of the usage probability distributions that control the selection process and has not proven to be a problem in current applications of the methods. In a study performed by Dyer on the level of requirements coverage using statistical testing, 100 per cent of the high-level requirements were covered, 90 per cent of the subcomponent-level requirements were covered, and approximately 80 percent of all requirements were covered[4].

The Cleanroom method asserts that statistical usage testing is many times more efficient than traditional coverage testing in improving the reliability of software. Statistical testing, which tends to find errors in the same order as their seriousness (from a user's point of view), will uncover failures several times more effectively than by randomly finding errors without regard to their seriousness. The basis for software reliability starts with the definition of a statsitical model, generally based on the concept that input data comes in at random times and with random contents. With defined initial conditions, any such fixed use is distinguishable from any other use. These uses can be assembled into a sequence of uses, and the collection identified as a stochastic process subject to evaluation using statistical methods.

Coverage testing is anecdotal and can only provide confidence about the specific paths tested. No assessment can be made about the paths not tested. Because usage testing exercises the software the way the users intend to use it, high-frequency errors tend to be found early. For this reason, statistical usage testing is more effective at improving software reliability than is coverage testing. Coverage testing is as likely to find a rare execution failure as it is to find a frequent one. If the goal of a testing program is to maximize the expected mean-time-to-failure, hence the reliability of the system, a strategy that concentrates on failures that occur more frequently is more effective than one that has an equal probability of finding high- and low-frequency failures[5].

Human functional verification has proven to be surprisingly synergistic with statistical testing according to Mills, Dyer, and Linger[1]. Experimental data from projects where both Cleanroom verification and more traditional debugging techniques were used show that the Cleanroom verified software exhibited fewer errors injected. Those errors were less severe and required less time to fix.

Software Process Maturity Factor. Using the Cleanroom methodology requires a change in paradigm--from viewing software development as an art or craft to viewing it as an engineering discipline. As such, it must have a rigorous foundation. In other engineering disciplines, failures are neither expected nor accepted as normal. Other engineering professions have minimized error by developing a sound theoretical base on which to build design practices. Cleanroom methods provide a theoretical foundation for a comprehensive engineering process that has been reduced to practice for commercial software development.

Using Cleanroom methods requires commitments from management to provide training (for both management and technical personnel) in the skills needed to implement the methodology. It also requires discipline. Management must allow the process to unfold naturally, technical personnel must rigorously follow the process. It may require additional tools, such as some automated support to develop the randomly generated test suites from the usage probability distributions.

In spite of these requirements, many have found that the return made the investments worthwhile. If being able to develop high- quality software within budget and on schedule are a concern, the Cleanroom methodology may be worth your taking the time to investigate. Peter Senge, in his book *The Fifth Discipline[7],* said that 30 years is the typical incubation period for a basic innovation, a concept that transforms an existing industry. Maybe we are now at the stage when the Clearoom methodology will begin to take hold and make that transformation of the software development industry.

Johnnie Henderson
Loral Space Information Systems
Software Technology Support Center
Ogden ALC/TISE
Hill AFB, UT 84056-5205
Voice: 801-777-8057 DSN 777-8057
Fax: 801-777-8069 DSN 777-8069
Internet: hendersj@software.hill.af.mil

About the Author

Johnnie Henderson is a consultant with Loral Space Information Systems in Houston, TX. She was the quality coordinator for the onboard shuttle software organization and leader of the project quality team. Her efforts as a member of the onboard shuttle software development organization helped create the software development process that was evaluated at a CMM level 5 by a team of NASA evaluators.

Ms. Henderson helped her team to institutionalize the oversight defect analysis process for the shuttle flight software. This activity produced significant process improvements for the software development process, resulting in the delivery of three software releases which have supported over fourteen shuttle missions with zero product defects.

References

1. Mills,H. D., M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software* 4, No. 5, 19-24 (September 1987).
2. Dyer, M., and A. Kouchakdjian, "Correctness Verification: Alternative to Structural Software Testing," *Information and Software Technololgy*, pp.53-59, Jan./Feb. 1990.
3. Hausler, P. A., R. C. Linger, and C. J. Trammel, "Adopting Cleanroom Software Engineering with a Phased Approach," *IBM Systems Journal* Vol. 33, No.1, 1994, pp. 89-109.
4. Dyer, M., *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, Inc., New York (1992).
5. Cobb, R. C., and H. D. Mills, "Engineering Software under Statistical Quality Control," *IEEE Software*, pp 44-54, November 1990.
6. Linger, R. C., "Cleanroom Process Model," *IEEE Software*, pp 50-58, March 1994.
7. Senge, P. M., *The Fifth Discipline: The Art and Practice of the Learning Organization*, Doubleday/Currency, New York, (1990).

LIST OF CLEANROOM PUBLICATIONS

Cobb, R. H., H. D. Mills, and A. Kouchakdjian, "The Cleanroom Engineering Software Development Process Manual," SET, 1991.

Deck, M. D., and P. A. Housler, "Cleanroom Software Engineering: Theory and Practice," *Proceedings of the Software Engineering and Knowledge Engineering Conference,* Skokie, IL, Knowledge Systems Institute, 1990.

Dyer, Michael, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, Inc. New York, 1994.

Green, S. E., and Rose Pajerski, "Cleanroom Process Evolution in the SEL," *Proceedings of 16th Annuall Software Engineering Workshop,* NASA Goddard Space Flight Center, Greenbelt, MD. 1991.

Hausler, P. A., "A Recent Cleanroom Success Story: The Redwing Project," *Proceedings of 17th Annual Software Engineering Workshop,"* NASA Goddard Space Flight Center, Greenbelt, MD. 1992.

Kouchakdjian, A., S. Green, and V. R. Basili, "Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory," *Proceedings of 14th Annual Software Engineering Workshop,* NASA Goddard Space Flight Center, Greenbelt, MD. 1989.

Linger, R. C., and H. D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility," *Proceeding of Compsac,* IEEE Computer Society Press, Los Alamitos, CA., 1988, pp. 10-17.

Mills, H. D., M. Dyer, and R. C. Linger, "Cleanroom Software Engineering," *IEEE Software*, Sept. 1988.

Spangler, A., and P. A. Housler, "The Cleanroom Software Engineering Process: An Overview," *Proceedings of the 3rd International Conference for Systems Integration,"* Sao Paulo, Brazil, IEEE Computer Society Press, Aug. 1994.

Tann, L. G., "OS32 and Cleanroom," *Proceedings of the 1st European Industrial Symposium on Cleanroom Software Engineering,* Q-labs, Lund, Sweden, 1993.

Trammell, C. J., L. H. Binder, and C. E. Snyder, "The Automated Production Control System: A Case Study in Cleanroom Software Engineering," *ACM Transations in Software Engineering and Methodology,* Jan. 1992, pp. 81-94.

LIST OF ORGANIZATIONS THAT PROVIDE CLEANROOM SERVICES

IBM Cleanroom Software Technology Center (CSTC)
100 Lake Forest Blvd.
Gaithersburg, MD 20877

POC: Philip Housler
Phone: 301-803-2684
internet: housler@vnet.ibm.com

Assessments:

Process Documentation:

Cleanroom Training:
- Cleanroom Overview
- Cleanroom System Development: Specification and Architecture
- Cleanroom Software Development: Design and Verification
- Cleanroom Certification.

Cleanroom Project Consulting:
- Cleanroom Project Planning and Project Management
- Cleanroom Specification Consulting
- Cleanroom Code Development and Verification Consulting
- Software Reliability Testing Consulting

High-Quality Cleanroom Software Development

Cleanroom Reliability Testing

Software Engineering Technology, Inc.
4600 Forbes Blvd.
Lanham, MD 20706

Phone: 301-731-6200
FAX: 301-731-6203

Services

Engineering Software Solutions:

Training:

Process Engineering:

Software Certification:

Research and Development:

Products

Cleanroom Engineering Handbooks:

Cleanroom Engineering Process Assistant:

The Certification Package:
- The Certification Assistant
- The Software Certifier's Handbook
- The Software Certifier's Training Program
- Certification Support Services

The following list was identified just prior to publication and we were not able to obtain a summary of services. Feel free to contact these organizations/people for additional information on Cleanroom services.

STARS Center
801 N. Randolph St.
Suite 400
Arlington, VA 22203
703-351-5300

Richard C. Linger
20221 Darlington Drive
Gaithersburg, MD 20879
301-926-4858